vip ENTERPRISE 8

ContentManager

Programmierhandbuch



http://www.gaussvip.com

Programmversion: 8.1.1

Dokumentenversion: De-04

Erscheinungsdatum: November 2002

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

KAPITEL 1

Einleitung

Das VIP Java API ermöglicht den Zugriff auf die Funktionen von VIP ContentManager über eine Java-basierte Programmierschnittstelle. Durch die Klassen und Interfaces des VIP Java API können Sie auf die von VIP ContentManager verwalteten Inhalte zugreifen und diese im Kontext der eigenen Softwareentwicklungen nutzen. Mögliche Anwendungen sind z.B. die Einbindung von Fremdsystemen oder die Überwachung von Aktionen, mit der Möglichkeit, die Durchführung von Aktionen zu unterbinden (Veto-Mechanismus).

1.1 Hinweise zu dieser Dokumentation

Diese Dokumentation richtet sich an Softwareentwickler, die über entsprechende Vorkenntnisse sowohl hinsichtlich der Funktionalität von VIP ContentManager als auch in der Programmiersprache Java verfügen. Die einzelnen Klassen, Interfaces und Methoden des VIP Java API werden hier nur kurz dargestellt. Vollständige, detaillierte Beschreibungen finden Sie in der Online-Dokumentation (Javadoc). Die Javadoc-Dokumentation befindet sich im Verzeichnis **\documentation\javadoc** im VIP-Installationsverzeichnis.

Die hier beschriebene Programmierschnittstelle ist Bestandteil der VIP CM Suite. Zusätzlich zum vorliegenden Programmierhandbuch können Sie Informationen aus folgenden Quellen beziehen:

- VIP ContentManager-Benutzerhandbuch: Dieses Dokument weist Sie ausführlich in alle Aufgaben der redaktionellen Pflege von Websites im VIP-CM-Workflow ein.
- VIP ContentManager-Administratorhandbuch: Dieses Dokument beschreibt die Installation, Konfiguration und Administration von VIP-CM-Systemen und enthält eine ausführliche Darstellung der technischen Konzepte von VIP ContentManager.

Die Inhalte dieses Handbuchs sind wie folgt gegliedert:

- Kapitel 2 “Konzepte” erläutert die grundlegenden Konzepte des VIP Java API.
- Kapitel 3 “Einsatz des VIP Java API” beschreibt, wie eigene Klassen in Form von Server-Agenten eingebunden werden.
- Das VIP Java API besteht mehreren Schnittstellen, die die folgenden Bereiche abdecken: Administration, Ereignisverarbeitung, Kontextverwaltung, Objektverwaltung, Deployment, Poolverwaltung und Systemverwaltung. Die dazu gehörigen Schnittstellen werden in den Kapiteln 4 bis 10 vorgestellt.
- Einige Beispiele für den Einsatz des VIP Java API finden Sie in Kapitel 11 “Anwendungsbeispiele”.

1.2 Neue Funktionen in der Version 8.1.1

Dieser Abschnitt bietet einen Überblick über die wesentlichen Änderungen im VIP Java API, die mit der Version 8.1.1 der VIP CM Suite eingeführt wurden.

Erweitertes `AdminHandler`-Interface

Das erweiterte `AdminHandler`-Interface bietet Zugriff auf Funktionen der Benutzer- und Systemverwaltung von VIP ContentManager. Mithilfe der Funktionen dieses Interfaces können Sie nun z.B. neue Benutzer anlegen, Zuordnungen von Principals vornehmen oder Funktionsbereiche anlegen. Ausführliche Informationen enthält Kapitel 4 “Administrationsinterface”.

Neues Interface `DeploymentHandler`

Das neue Interface `DeploymentHandler` bietet Zugriff auf die Metadaten von Web-Objekten (URL und Pfad) sowie Informationen zum Status und zu Fehlern des Deployments. Ausführliche Informationen enthält Kapitel 8 “Deployment”.

Neues Interface `SystemHandler`

Das neue Interface `SystemHandler` bietet Methoden zum Zugriff auf die Bericht-, Protokoll- und Tracing-Funktionen der VIP-CM-Server. Außerdem können die am VIP-CM-System angemeldeten Benutzer ermittelt werden. Ausführliche Informationen enthält Kapitel 10 “Systemverwaltung”.

> **Hinweis:** Einen Überblick über Methoden, die mit der Version 8.1.1 der VIP CM Suite auf “deprecated” gesetzt wurden, enthält die Online-Dokumentation (Javadoc).

1.3 Typographische Konventionen

Programmelemente u.Ä. werden im Text folgendermaßen hervorgehoben:

Element	Schriftart oder Symbol	Beispiele
Programmoberfläche wie z.B. Menübefehle, Fenster, Dialoge, Feld- und Schaltflächen-bezeichnungen	*Menü ⟶ Eintrag*	*Datei ⟶ Anlegen*
Pfade zu Verzeichnis-sen, Namen von Dateien und Ver-zeichnissen	**Laufwerk:\Verzeichnis\Dateiname**	**D:\VIP8**

Zitate aus Programm-code oder Konfigurationsdateien	`Code-Zitate`	`<head>`
		`<title>{VIPTITLE}</title>`
		`</head>`

Variablen, d.h. Platzhalter für bestimmte Elemente	{Variable}	{VIP-Installationsver-zeichnis}

Wichtige **Hinweise** und **Warnungen** stehen in grauen Kästen. Diese Informationen sollten Sie unbedingt lesen, um Fehler bei der Nutzung und Verwaltung von VIP-CM-Systemen sowie Datenverluste zu vermeiden.

KAPITEL 2

Konzepte

Dieses Kapitel beschreibt in kurzer Form die grundlegenden Konzepte von VIP ContentManager. Dabei steht die Beschreibung der Begriffe im Vordergrund, die beim Arbeiten mit VIP ContentManager eine Rolle spielen.

Eine ausführliche Beschreibung der technischen Konzepte sowie der Architektur von VIP ContentManager finden Sie im VIP ContentManager-Administratorhandbuch. Weitere Informationen zum Arbeiten mit einem VIP-CM-System sind im VIP ContentManager-Benutzerhandbuch enthalten.

2.1 Objektverwaltung

Das VIP Java API bietet Zugriff auf wesentliche Methoden der Objektverwaltung. Die Bearbeitung von VIP-Objekten unterliegt einem fest definierten Workflow aus Editieren (Bearbeiten), Qualitätssicherung und Veröffentlichung im Produktionsbetrieb. Welche Aktionen an einem VIP-Objekt durchgeführt werden können, ist vom Status des Objekts im Workflow, dem Objekttyp und den Zugriffsrechten des angemeldeten Benutzers abhängig. Die Zusammenhänge zwischen diesen Aspekten sollen in den folgenden Abschnitten erläutert werden.

Objektdaten

VIP ContentManager dient der Verwaltung von Websites. Websites enthalten bestimmte Dokumente, die für die verschiedensten Zielgruppen verwaltet werden. Solche Dokumente können z.B. HTML- oder XML-Dokumente, Grafiken oder Microsoft Word-Dokumente sein. Über einen Webserver können Sie im Inter-, Intra- oder Extranet veröffentlicht werden.

Die Dokumente einer von VIP ContentManager verwalteten Website werden als VIP-Objekte bezeichnet. Jedes VIP-Objekt hat einen bestimmten Typ (siehe Abschnitt “Objekttyp” auf Seite 20) und setzt sich aus folgenden Bestandteilen, den so genannten Objektdaten, zusammen:

- *Inhalt:* Der Inhalt (Content) eines VIP-Objekts sind die Daten des eigentlichen Dokuments wie z.B. die Zeichenfolge in einer HTML-Datei oder die Folge von Bytes einer Grafikdatei.
- *Metadaten:* Die Metadaten eines Objekts liefern Informationen über das Dokument, z.B. Erstellungsdatum, Autor, Titel, Objekttyp.
- *Protokoll:* Das Protokoll liefert eine Beschreibung sämtlicher Änderungen, die am Objekt durchgeführt wurden.

Sämtliche VIP-Objekte werden über ihre OID (*Object Identifier*) referenziert. Ohne Angabe der Version identifiziert die OID das *aktuelle* VIP-Objekt. Um ältere Versionen zu erhalten, wird zusätzlich eine Versionsangabe benötigt.

Neben einem definierten Satz von Metadaten können Sie jedem Objekt über seinen Objekttyp eine Attributmenge zuordnen. Eine Attributmenge ist eine Menge von Attributen, die spezielle Eigenschaften von Objekttypen beschreiben. Dazu können u.a. die Auflösung von Grafiken oder ein Copyright-Vermerk gehören.

Außerdem können Sie die Inhalte von VIP-Objekten kategorisieren. Zu diesem Zweck definieren Sie Objektkategorien, für die Sie eine Reihe von Eigenschaften festlegen können. Eine mögliche Objektkategorie wären Rechnungen, die durch die Eigenschaften Rechnungsempfänger und Status gekennzeichnet sein können.

Ausführliche Informationen zur Bearbeitung von VIP-Objekten über die Funktionen des VIP Java API bietet Kapitel 7 “Objektverwaltung”.

Objektstatus

Die Erstellung und Bearbeitung von VIP-Objekten mit VIP ContentManager unterliegt einem fest definierten Workflow. Durch die verschiedenen Arbeitsschritte befindet sich ein VIP-Objekt immer in einem bestimmten Bearbeitungszustand. Der Workflow von VIP ContentManager bestimmt somit den Status eines VIP-Objekts. Entsprechend den Aktionen der Objektverarbeitung werden die folgenden Objektstatus unterschieden:

- geändert (auch bei neu angelegten Objekten)
- ausgeliehen
- vorgelegt
- abgelehnt
- freigegeben
- verzögert freigegeben
- gelöscht

Die Objektstatus werden im VIP Java API durch das Interface `ObjectState` repräsentiert, siehe Abschnitt “Objektstatus – Konstanten und mögliche Aktionen” auf Seite 143.

Durch die verschiedenen Arbeitsschritte bzw. Aktionen am VIP-Objekt ändert sich der jeweilige Status eines Objekts. Jede Statusänderung wird protokolliert. Dabei wird die vorherige Version eines VIP-Objekts getrennt von der aktuellen Version archiviert, sodass ältere Versionen nicht verloren gehen. Auf diese Versionen können Sie jederzeit wieder zugreifen.

Welche Aktionen an einem VIP-Objekt durchgeführt werden können, hängt von folgenden Faktoren ab:

- der Datenhaltungssicht (Edit, QS oder Produktion)
- dem aktuellen Status des VIP-Objekts

  > **Hinweis:** Die Möglichkeit, bestimmte Aktionen am Objekt durchzuführen, hängt auch vom Status des übergeordneten Themas innerhalb der Navigationstopologie ab.

- der Zugriffssteuerungsliste des VIP-Objekts
- den zugewiesenen Funktionsbereichen des Benutzers

Objekttyp

Der Objekttyp bestimmt, zu welcher Klasse von Dokumenten ein VIP-Objekt gehört. Daraus ergeben sich verschiedene Eigenschaften des VIP-Objekts, u. a.:

- Attribute aus einer Attributmenge, die das VIP-Objekt besitzen kann
- Art der Vorlagen, die das VIP-Objekt verwenden kann
- Dateien, die bei der Seitengenerierung als Repräsentation des VIP-Objekts erzeugt werden können

Neben Objekttypen wie z.B. “HTML-Seite”, “JSP-Vorlage” oder “PDF-Dokument” gibt es Themenobjekte. Ein Themenobjekt kann andere untergeordnete Objekte aufnehmen. Es dient insofern auch der Strukturierung der VIP-Objekte in der Navigationssicht und damit der Strukturierung der Website. Ein weiterer wichtiger Objekttyp sind Vorlagenobjekte. Sie bestimmen das Layout und sind für die Seitengenerierung wesentlich.

Die Objekttypen werden im VIP Java API durch das Interface ObjectType repräsentiert, siehe Abschnitt “Objekttypen” auf Seite 149.

Workflow und Datenhaltungssichten

Entsprechend dem Bearbeitungsstatus eines VIP-Objekts gibt es verschiedene Sichten auf das Objekt: Edit-, QS- und Produktionssicht.

- *Edit-Sicht:* Die Edit-Sicht repräsentiert den Arbeitsschritt der Erstellung und Bearbeitung von VIP-Objekten. Die Objekte wie beispielsweise HTML-Seiten werden von Redakteuren oder Grafikern angelegt und geändert. Dafür benötigen diese Benutzer entsprechende Zugriffsrechte für die Objekte. Nach der Bearbeitung werden die Objekte der Qualitätssicherung vorgelegt. Dadurch werden die Änderungen am Objekt auch in der QS-Sicht sichtbar.
- *QS-Sicht:* Die QS-Sicht zeigt die VIP-Objekte einschließlich aller Änderungen, die der Qualitätssicherung vorgelegt worden sind. Mitarbeiter der Qualitätssicherung können die Änderungen inhaltlich und formal prüfen. Aufgrund dieser Prüfung wird entschieden, ob ein Objekt zur Nachbesserung zurückgeschickt oder freigegeben wird. Für die Freigabe ist das entsprechende Zugriffsrecht nötig. Durch die Freigabe werden die Objekte in die Produktionssicht übertragen. Damit wird die aktuelle Objektversion in der publizierten Website verfügbar.

- *Produktionssicht*: Diese Sicht stellt die freigegebenen Seiten einer Website bereit. Mithilfe eines HTTP-Servers kann auf die Seiten über das Internet oder Intranet zugegriffen werden.

Je nach Objektstatus sind die VIP-Objekte einer Website in verschiedenen Sichten verfügbar und können bearbeitet werden. Ein neu angelegtes Objekt ist beispielsweise nur in der Edit-Sicht sichtbar, nicht aber in der QS- oder Produktionssicht. Durch verschiedene Aktionen im Rahmen der Objektbearbeitung kann der Status eines VIP-Objekts in den verschiedenen Sichten unterschiedlich sein.

Die folgende Grafik veranschaulicht den Workflow bei der Bearbeitung von VIP-Objekten:

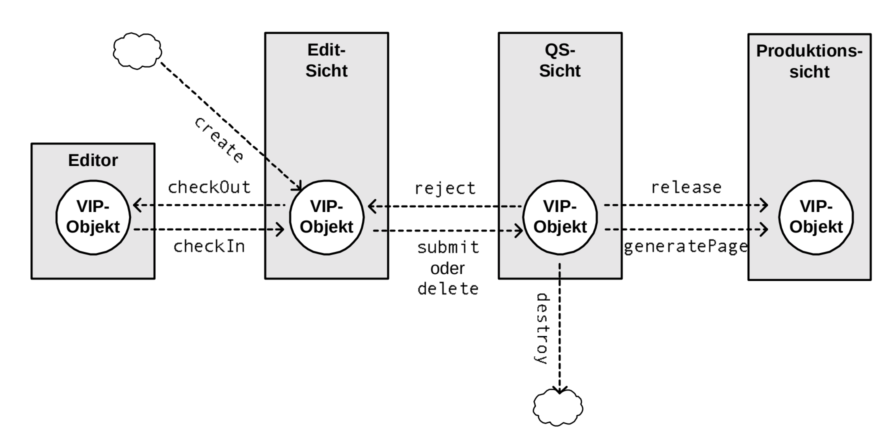


Abb. 1 – Workflow bei der Bearbeitung eines VIP-Objekts

Verfügbarkeit der Sichten auf den VIP-CM-Servern

Die verschiedenen Sichten auf die VIP-Objekte werden durch die Deploymentsysteme erzeugt, dabei bestimmt der Typ des Deploymentsystems die Sicht: Ein Edit-Deploymentsystem erzeugt Web-Objekte in der Edit-Sicht usw.

Welche Typen von Deploymentsystemen auf einem VIP-CM-Server installiert werden können, ist wiederum vom *Servertyp* abhängig (Edit, QS, Produktion, PortalManager). Auf einem Edit-Server können Deploymentsysteme für alle Sichten installiert werden, sodass auf diesem Servertyp der gesamte VIP-CM-Workflow zur Verfügung steht. Auf einem QS-Server können hingegen nur QS- und Produktionsdeploymentsysteme installiert werden.

Die folgende Tabelle veranschaulicht die Verfügbarkeit der verschiedenen Datenhaltungssichten auf den Servertypen:

Tabelle 1 – Servertypen und mögliche Datenhaltungssichten

Servertyp	Edit-Sicht	QS-Sicht	Produktionssicht
Edit	✓	✓	✓
QS		✓	✓
Produktion			✓
PortalManager*	✓	✓	✓

* Die möglichen Datenhaltungssichten auf einem PortalManager-Server hängen davon ab, von welchem anderen VIP-CM-Server dieser benachrichtigt wird (Routing-Einstellung der Website). Erhält der PortalManager-Server seine Daten von einem Edit-Server, können alle drei Typen von Deploymentsystemen installiert werden.

Beim Zugriff auf VIP-Objekte muss die verwendete Datenhaltungssicht angegeben werden. Dazu dienen die Typkonstanten im Interface `Server`, siehe Abschnitt “Datenhaltungssichten” auf Seite 147.

Neben der Einteilung in verschiedene Servertypen werden die Server des VIP-CM-Systems außerdem in *Kategorien* (Master- und Proxy-Server) unterteilt. In jedem VIP-CM-System gibt es einen Master-Administrationsserver zur Verwaltung der Konfigurations- und Systemdaten und einen oder mehrere Master-Edit-Server zur Verwaltung der Website-Daten. Physikalische Änderungen an den VIP-Objekten können nur Master-Edit-Server durchführen. Zusätzlich dazu können Sie beliebig viele Proxy-Server einrichten. Die Proxy-Server delegieren entsprechende Aktionen an den Master-Edit-Server der Website.

Informationen zu VIP-CM-Servern können über das Interface `Server` ermittelt werden, siehe Abschnitt 4.6 “Informationen zu VIP-CM-Servern” auf Seite 96.

Topologien – Organisation von VIP-Objekten

Die VIP-Objekte einer Website können aufgrund bestimmter Eigenschaften (Metadaten) als hierarchische Struktur mit über- und untergeordneten Objekten dargestellt werden. Im Interface `Topology` sind die folgenden Topologien, auf Basis verschiedener Prinzipien der hierarchischen Organisation von VIP-Objekten, definiert:

Themenstruktur (Navigationstopologie)

In der Themenstruktur werden Objekte Themen zugewiesen. Jedes Thema ist wiederum ein VIP-Objekt, das einem anderen Thema untergeordnet ist. Diese Hierarchie ist vergleichbar mit Verzeichnissen und Unterverzeichnissen in einem Dateisystem.

Für jede Website gibt es genau ein Thema, welches das Wurzelobjekt des Navigationsbaums einer Website ist. Ausgehend vom Wurzelobjekt verzweigt der Baum in Themen und Unterthemen. Alle Objekte sind diesem Wurzelobjekt direkt oder indirekt untergeordnet.

Vorlagenstruktur (Vorlagentopologie)

In der Vorlagenstruktur werden Objekte nach ihren Vorlagen geordnet. Vorlagen sind ebenfalls VIP-Objekte. Jede Website kann eine oder mehrere Vorlagen enthalten, die ineinander verschachtelt sein können. Alle Objekte werden ihren jeweiligen Vorlagen untergeordnet. Objekte, die keine Vorlagen besitzen und selbst nicht vom Objekttyp “Vorlage” sind, werden in dieser Sicht nicht dargestellt.

2.2 Zugriffssteuerung

Durch die Funktionen der Zugriffssteuerung kann mit VIP ContentManager der Zugriff auf die Objekte einer Website und auf die Administration des VIP-CM-Systems genau gesteuert werden.

Die Rechtestruktur in einem VIP-CM-System wird durch mehrere Komponenten gebildet:

- Die Zugehörigkeit von Benutzern zu Gruppen und Rollen

  Rollen werden zumeist aufgabenbezogen definiert, während Gruppen in der Regel an organisatorische Strukturen wie Abteilungen oder Projekte geknüpft sind.

  Benutzer, Gruppen und Rollen werden als *Principals* bezeichnet.

- Die Zuweisung von Funktionsbereichen zu Principals

  Die Funktionsbereiche steuern, welche Objekte die Benutzer anlegen dürfen und welche Ansichten, Dialoge und Funktionen ihnen im CMS-Client zur Verfügung stehen.

- Das Festlegen von Standard-Objektrechten für Principals

  Für die Principals können Vorgaben für die Objektrechte definiert werden. Über die Objektrechte wird festgelegt, welche Aktionen dem Benutzer für ein VIP-Objekt, dem er zugeordnet wurde, standardmäßig zur Verfügung stehen (Lesen, Metadaten ändern, Freigeben etc.).

- Das Festlegen von Zugriffsrechten für die Objekte der mit VIP verwalteten Websites

  Für die einzelnen Objekte der Website kann in einer Zugriffssteuerungsliste festgelegt werden, welche Gruppen und Rollen (bzw. einzelne Benutzer) Zugriff erhalten und welche konkreten Objektrechte für den Zugriff gelten. In diesem Prozess können die Standard-Objektrechte der Principals, die Zugriff auf ein VIP-Objekt haben, geändert werden.

- Das Festlegen der Administrationsrechte für das VIP-CM-System

  Jeder Principal kann Administrationsrechte für das VIP-CM-System erhalten, die abgestuft vergeben werden können. Nur Principals mit Administrationsrechten können Konfigurationsänderungen vornehmen oder den Systemstatus ändern.

Profile von Benutzern, Gruppen und Rollen

Die Profile der Principals enthalten folgende Einstellungen:

- Eigenschaften wie E-Mail-Adresse, Kennung oder Name
- Zuordnungen zu anderen Principals (z. B. von Benutzern zu Gruppen/Rollen und Stellvertreterzuordnungen), zu Websites und Funktionsbereichen
- Administrationsrechte und Standard-Objektrechte

Mithilfe der Funktionen des VIP Java API können Benutzer angelegt und gelöscht sowie deren Profile bearbeitet werden. Ausführliche Informationen dazu enthält das Kapitel 4 “Administrationsinterface”.

Funktionsbereiche

Jedem Principal können Funktionsbereiche zugewiesen werden. Über die Funktionsbereiche steuern Sie, welche Funktionen von VIP ContentManager dem Benutzer zur Verfügung stehen. Dabei erfüllen die Funktionsbereiche zwei wesentliche Aufgaben:

1. Sie legen fest, welche Typen von Objekten die Benutzer anlegen dürfen. Die Funktionsbereiche wie z. B. “Anlegen Basis” sind standardmäßig mit Objekttypen verknüpft. Nur Benutzer, die über den entsprechenden Funktionsbereich verfügen, können Objekte mit diesem Objekttyp anlegen.

   Die im VIP-Administrationsprogramm vorgegebenen Verknüpfungen mit Objekttypen können dort auch geändert werden.

2. Sie bestimmen, welche Ansichten und Dialoge dem Benutzer im CMS-Client zur Verfügung stehen. Auf diese Weise wird festgelegt, welche Aktionen der Benutzer durchführen kann. So wird z. B. der Zugriffsrechte-Dialog nur angezeigt, wenn der entsprechende Benutzer über den Funktionsbereich “Dialog Zugriffsrechte” verfügt.

Über die Funktionen des VIP Java API können

- Funktionsbereiche angelegt und gelöscht werden (siehe Abschnitt 4.4 “Funktionsbereiche bearbeiten” ab Seite 89)
- Zuordnungen zwischen Funktionsbereichen und Principals hergestellt und aufgehoben werden (siehe Abschnitt 4.5 “Zuordnungen von Principals bearbeiten” auf Seite 94)

Zugriffssteuerungsliste

VIP ContentManager führt für jedes VIP-Objekt eine Zugriffssteuerungsliste (*Access Control List = ACL*). In dieser Liste ist festgelegt, über welche Zugriffsrechte die jeweiligen Benutzer, Gruppen oder Rollen verfügen.

Wird ein Principal der Zugriffssteuerungsliste hinzugefügt, werden zunächst die Standard-Objektrechte des Principals gesetzt (siehe “Standard-Objektrechte von Principals” auf Seite 86). Diese Rechte können auf Objektebene jederzeit geändert werden. Dabei können Zugriffsrechte explizit als erlaubt bzw. verboten ausgezeichnet werden, oder sie werden nicht explizit gesetzt. Bei einer Rechteüberprüfung werden im letzteren Fall bei Benutzern alle zugewiesenen Gruppen und Rollen dahingehend untersucht, ob sie eine explizite Erlaubnis oder ein Verbot definieren (wobei ein Verbot immer Vorrang hat).

Die Zugriffssteuerungslisten werden vererbt, d.h., jedes in der Navigationssicht tiefer angeordnete VIP-Objekt erbt in der Regel die Rechte des übergeordneten Objekts (des Vaters). Beim Anlegen einer Website wird ein Principal ausgewählt, der initial vollen Zugriff auf alle Objekte der Website hat. Dieser wird in die Zugriffssteuerungsliste des Wurzelobjekts der Website aufgenommen. Alle weiteren VIP-Objekte dieser Website erben, falls nicht explizit gesetzt, die Rechte des Wurzelobjekts.

Hinweise zur Bearbeitung der Zugriffssteuerungsliste von VIP-Objekten über das VIP Java API erhalten Sie im Abschnitt “Zugriffssteuerungslisten – das Interface ACL” ab Seite 206.

Administrationsrechte für das VIP-CM-System

Neben den Zugriffsrechten für VIP-Objekte gibt es Administrationsrechte für das VIP-CM-System. Principals mit Administrationsrechten haben Zugriff auf die Konfiguration, Benutzer- und Systemverwaltung sowie die Systemübersicht von VIP ContentManager. Die Administrationsrechte können abgestuft gewährt werden, sodass ein Benutzer nur Teilbereiche der Administration einsehen oder bearbeiten kann.

Um bestimmte Funktionen des VIP Java API auszuführen, benötigt der Benutzer, der für die Anmeldung eines Agenten verwendet wird, Administrationsrechte. So können Benutzer nur angelegt werden, wenn das Administrationsrecht “Principal anlegen, ändern, löschen” vorhanden ist.

Einige der Rechte können nur zusammen mit anderen vergeben werden, d. h., sie beinhalten andere Rechte. Wenn Sie z. B. einem Principal das Recht “Konfigurationseintrag anlegen, ändern, löschen” gewähren, erhält er automatisch auch die Rechte “Zugriff auf Konfiguration” und “Konfigurationseintrag ändern”.

Die folgende Tabelle gibt einen Überblick über die einzelnen Rechte und deren Abhängigkeiten:

Tabelle 2 – Die einzelnen Administrationsrechte

Recht	Mögliche Aktionen
Lesender Zugriff auf Benutzerverwaltung (API)	Wenn ein Server-Agent die Benutzerinformationen lesen soll, muss der Benutzer, der für die Anmeldung des Agenten verwendet wird, über dieses Recht verfügen.
Zugriff auf Benutzerverwaltung	Lesender Zugriff auf die Benutzerinformationen Die Baumelemente *Benutzer*, *Gruppen* und *Rollen* in der Ansicht *Benutzerverwaltung* werden nur eingeblendet, wenn der Benutzer dieses Recht hat.
Principal ändern	Ändern der Einstellungen von Benutzern, Gruppen und Rollen sowie der Zuordnungen zwischen diesen Elementen, z. B. die Zuordnung eines Benutzers zu einer Gruppe. Um die Zuordnung von Principals zu Funktionsbereichen und Websites ändern zu können, ist außerdem das Recht “Zuordnung zu Website/Funktionsbereich ändern” erforderlich. Beinhaltet das Recht “Zugriff auf Benutzerverwaltung”
Zuordnung zu Website/ Funktionsbereich ändern	Ändern der Zuordnung von Benutzern, Gruppen und Rollen zu Websites und Funktionsbereichen Beinhaltet das Recht “Zugriff auf Benutzerverwaltung”
Principal anlegen, ändern, löschen	Anlegen, Bearbeiten und Löschen von Benutzern, Gruppen und Rollen sowie Einrichten bestehender Principals aus einem LDAP-Verzeichnisdienst als VIP-Principals Beinhaltet die Rechte “Principal ändern”, “Zugriff auf Benutzerverwaltung” und “Zuordnung zu Website/Funktionsbereich ändern”

Recht	**Mögliche Aktionen**
Administrationsrechte ändern	Bearbeiten der Administrationsrechte von Principals Beinhaltet das Recht “Zugriff auf Benutzerverwaltung”
Zugriff auf Konfiguration	Lesender Zugriff auf die Konfiguration von VIP ContentManager Die Baumelemente *Funktionsbereiche* und *Websites* der Ansicht *Benutzerverwaltung* sowie die gesamte Ansicht *Konfiguration* sind nur sichtbar, wenn der Benutzer dieses Recht hat.
Konfigurationseintrag ändern	Bearbeiten von Elementen in der Ansicht *Konfiguration* einschließlich der Zuordnungen zwischen den Elementen, z.B. die Zuordnung von Proxy-Servern zu Websites Beinhaltet das Recht “Zugriff auf Konfiguration”
Konfigurationseintrag anlegen, ändern, löschen	Anlegen, Bearbeiten und Löschen von Elementen in der Ansicht *Konfiguration*, z.B. von Websites oder Deploymentsystemen Beinhaltet die Rechte “Konfigurationseintrag ändern” und “Zugriff auf Konfiguration”
Zugriff auf Systemverwaltung	Lesender Zugriff auf die Systemverwaltung und auf die Systemübersicht Die Ansichten *Systemverwaltung* und *Systemübersicht* erscheinen nur, wenn der Benutzer dieses Recht hat.
Systemstatus ändern	Benutzer abmelden, Runlevels von Servern und Websites ändern und laufende Aktionen auf einem Server abbrechen, Befehle im Menü *Extras* nutzen Beinhaltet das Recht “Zugriff auf Systemverwaltung”

Die Administrationsrechte von Principals können über das `AdminHandler`-Interface bearbeitet werden, siehe Abschnitt 4.3 “Rechte von Principals bearbeiten” ab Seite 82.

2.3 Seitengenerierung (Deployment)

Für die Anzeige der Objekte in einem Browser wird aus dem VIP-Objekt ein so genanntes *Web-Objekt* generiert. Dieser Vorgang wird als Seitengenerierung bezeichnet.

Jedes VIP-Objekt kann eine Vorlage (*Template*) haben, die in der Regel ein Layout für die zu generierende Webseite definiert. Bei der Seitengenerierung wird der Inhalt des VIP-Objekts mit der Vorlage verbunden. Eine Vorlage ist selbst ein VIP-Objekt, das wiederum eine Vorlage enthalten kann. Die generierte Seite hängt u.a. von dieser Vorlagenhierarchie (*Kaskade*) ab.

Weiterhin spielen der Objekttyp des zu generierenden VIP-Objekts und die Objekttypen der Vorlagen eine wesentliche Rolle für die Seitengenerierung. So werden z.B. für Microsoft Word-Dokumente standardmäßig zwei Dateien erzeugt: das Dokument selbst und eine HTML-Seite mit einem Verweis auf die Word-Datei. Bei HTML-Objekten mit zugeordneter Vorlage wird der `<body>`-Bereich des HTML-Dokuments ausgeschnitten.

Entscheidend ist auch die Datenhaltungssicht, die bestimmt, welche Sicht auf die VIP-Objekte für die Generierung verwendet werden (Edit, QS oder Produktion), siehe Abschnitt “Workflow und Datenhaltungssichten” auf Seite 21.

Für die Seitengenerierung sind die *Deploymentsysteme* zuständig. Sie erzeugen aus den in der Datenbank gespeicherten VIP-Objekten die Web-Objekte, die mithilfe eines HTTP-Servers in einem Browser dargestellt werden können. Ein Deploymentsystem generiert die Web-Objekte für einen VIP-CM-Server, eine Website und eine Datenhaltungssicht. In einem VIP-CM-System können Sie beliebig viele Deploymentsysteme konfigurieren.

Jedes Web-Objekt wird über eine URL referenziert. Da für ein und dasselbe VIP-Objekt von mehreren Deploymentsystemen verschiedene Web-Objekte erzeugt werden können, gibt es im Allgemeinen mehrere URLs für ein und dasselbe VIP-Objekt.

Um auf die Daten von Deploymentsystemen zuzugreifen, können die Funktionen des Interfaces `DeploymentHandler` genutzt werden, siehe Kapitel 8 “Deployment”.

2.4 Authentifizierung und Kontextverwaltung

Für das Arbeiten mit dem VIP-CM-System muss sich ein Benutzer authentifizieren. Bei der Anmeldung ist daher die Angabe einer eindeutigen Benutzerkennung (*User-ID*) und eines Passworts erforderlich. Die Überprüfung der Angaben übernimmt ein Master- bzw. Proxy-Admin-Server.

Bei einer erfolgreichen Anmeldung gibt das VIP-CM-System eine so genannte Kontext-ID zurück, die die aktuelle Session für diesen Benutzer repräsentiert. Die Kontext-ID wird oft als Argument an API-Methoden übergeben, damit das System die Berechtigung für die von den Methoden ausgelösten Aktionen überprüfen kann.

Informationen zur Kontextverwaltung sowie zur An- und Abmeldung bietet Kapitel 6 “Kontextverwaltung”.

2.5 Ereignisse

In einem VIP-CM-System werden Ereignisse gefeuert, wenn sich der Status eines VIP-Objekts ändert, ein Web-Objekt verarbeitet wird oder sich der Zustand des gesamten Systems ändert. Dazu gehören sowohl Ereignisse, die nach der Aktion gefeuert werden (wenn die Statusveränderung erfolgt ist), als auch Vorbereitungsereignisse, die vor der eigentlichen Aktion gefeuert werden.

Agenten können sich auf Ereignisse registrieren. Bei der Registrierung auf Vorbereitungsereignisse kann eine Aktion durch das Einlegen eines Vetos (in Form einer `Exception`) abgebrochen werden. Ausführliche Informationen zu den Events des VIP Java API enthält Kapitel 5 “Ereignisverarbeitung”. Ein Beispiel für einen Agenten, der sich auf ein Ereignis registriert und gegebenenfalls eine Aktion durch ein Veto abbricht, finden Sie im Abschnitt 11.4 “Veto gegen Metadatenänderung” auf Seite 251.

2.6 Systemverwaltung

Neben der Objektverwaltung bietet das VIP Java API auch Zugriff auf die Funktionen der Systemverwaltung. So können die Bericht- und Tracing-Funktionen der VIP-CM-Server genutzt und Runlevels von Servern und Websites gesetzt werden. Der Zugriff auf diese Funktionen erfolgt über die Interfaces `SystemHandler` und `AdminHandler`:

- Interface `SystemHandler`: Ermitteln angemeldeter Benutzer sowie Zugriff auf Berichte, Protokolle und Tracing-Funktionen der VIP-CM-Server

  Siehe Kapitel 10 “Systemverwaltung”

- Interface `AdminHandler`: Ermitteln von Informationen über VIP-CM-Server, Websites und Deploymentsysteme sowie Ändern des Runlevels von Servern und Deploymentsystemen

  Siehe Kapitel 4 “Administrationsinterface”

KAPITEL 3

Einsatz des VIP Java API

Dieses Kapitel erläutert die Einbindung von Server-Agenten, die auf Grundlage des VIP Java API entwickelt wurden, in das VIP-CM-System. Außerdem wird die zentrale Klasse `VipRuntime` vorgestellt und die Fehlerbehandlung des VIP Java API beschrieben.

3.1 Server-Agenten

Server-Agenten sind Java-Implementationen auf Basis des VIP Java API. Durch Server-Agenten kann die Funktionalität der VIP-CM-Server ergänzt bzw. erweitert werden. Server-Agenten werden beim Starten eines VIP-CM-Servers geladen und in derselben Java Virtual Machine (JVM) wie der jeweilige VIP-CM-Server ausgeführt. Daher müssen alle Agenten auf Basis des zum Ausführen der VIP-CM-Server verwendeten Java SDK (Software-Development-Kit) entwickelt werden.

> **Hinweis:** Damit ein Server-Agent korrekt geladen wird, packen Sie die entsprechenden Klassen in eine JAR-Datei und kopieren Sie diese in das Verzeichnis **`\external_lib\`** im VIP-Installationsverzeichnis. Starten Sie anschließend den entsprechenden VIP-CM-Server neu.

Konfiguration von Parametern und Konstruktoren

Das Einbinden und die Konfiguration von Server-Agenten erfolgen im VIP-Administrationsprogramm über *Konfiguration* → *Server-Agenten*.

Hier definieren Sie die benötigten Eigenschaften des Agenten und setzen entsprechende Werte. Diese Eigenschaften können über einfache Parameter definiert werden, denen im VIP-Administrationsprogramm Werte zugewiesen werden. Es ist aber auch möglich, baumartige Hierarchien von Parametern zu definieren. Zu diesem Zweck legen Sie so genannte Knoten an. Ein Knoten kann einzelne Parameter oder weitere Unterknoten enthalten.

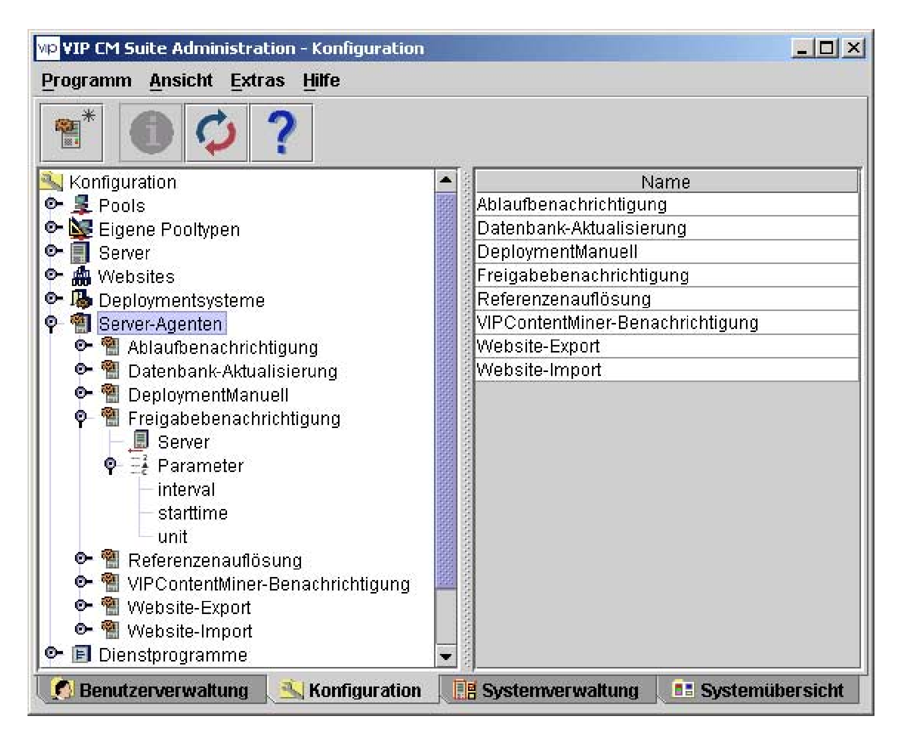


Abb. 2 – Server-Agenten im Admin-Client verwalten

Jeder Server-Agent muss das Interface `ServerAgent` (Package `de.gauss.vip.api`) implementieren. Der Agent benötigt immer einen als `public` deklarierten Konstruktor. Dieser sollte als Argument eine `de.gauss.vip.config.PropertyMap` erhalten. Das Interface `de.gauss.vip.config.PropertyMap` ist in der Javadoc zum API von VIP PortalManager (VIP Enterprise Objects) beschrieben.

Es besteht auch die Möglichkeit, einen Konstruktor mit einem Argument vom Typ `java.util.Properties` für den Agenten zu verwenden. Falls die Parameter des Agenten jedoch nicht flach, sondern hierarchisch mit Knoten organisiert sind, müssen Sie in jedem Fall ein Argument vom Typ `de.gauss.vip.config.PropertyMap` benutzen.

Der entsprechende Konstruktor wird beim Anlegen des Agenten-Objekts mit den in der Konfiguration eingestellten Eigenschaften aufgerufen. Die Eigenschaften werden mit der Methode `getProperty` bzw. `getPropertyValue` gelesen. In den Einstellungen eines Agenten wird auch die über den Klassenpfad zu erreichende Java-Klasse des Agenten angegeben.

Beispiel 1 – Agent mit hierarchischer Anordnung der Argumente

Die folgende Abbildung zeigt die Konfiguration eines Agenten, dessen Argumente in Knoten zusammengefasst sind.

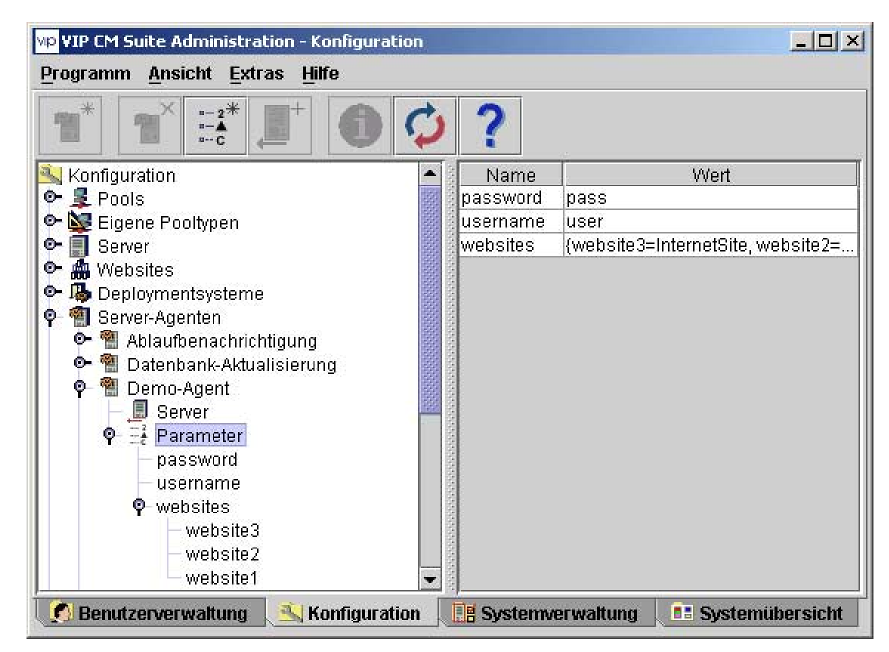


Abb. 3 – Konfiguration des Demo-Agenten mit Knoten

Der Konstruktor der Klasse `DemoAgent` mit geschachtelten Parametern sieht folgendermaßen aus:

```
public class DemoAgent
    implements de.gauss.vip.api.ServerAgent
{
    [...]
    private final String username;
    private final String password;
    private final String website1;
    private final String website2;
    private final String website3;

    public DemoAgent(de.gauss.vip.config.PropertyMap config)
        throws de.gauss.vip.config.KeyNotFoundException
    {
        de.gauss.vip.config.PropertyMap websites
            = config.getPropertyMap("websites");
        username = config.getPropertyValue("username");
```

```
        password = config.getPropertyValue("password");
        website1 = websites.getPropertyValue("website1");
        website2 = websites.getPropertyValue("website2");
        website3 = websites.getPropertyValue("website3");
    }
    [...]
}
```

Beispiel 2 – Agent mit Argumenten ohne Knoten

Die Konfiguration des Demo-Agenten ohne Knoten könnte folgendermaßen aussehen:

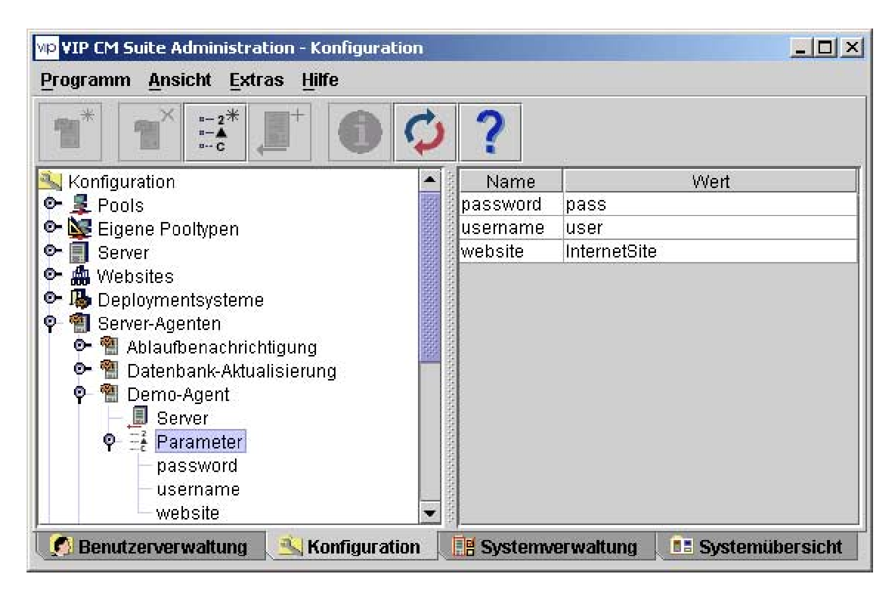


Abb. 4 – Konfiguration des Demo-Agenten ohne Knoten

Der Konstruktor der Klasse `DemoAgent` sieht wie folgt aus.

```
public class DemoAgent
    implements de.gauss.vip.api.ServerAgent
{
    [...]
    private final String userName;
    private final String password;
    private final String website;

    public DemoAgent(java.util.Properties config)
    {
        userName = config.getProperty("username");
        password = config.getProperty("password");
        website1 = config.getProperty("website");
    }
    [...]
}
```

Das Interface `ServerAgent`

Konstruktor

Jede Klasse, die als Server-Agent genutzt werden soll, muss einen öffentlichen (`public`) Konstruktor bereitstellen. Dieser Konstruktor muss entweder

- einen Parameter vom Typ `de.gauss.vip.config.PropertyMap` oder
- einen Parameter vom Typ `java.util.Properties` erwarten

Ist ein Konstruktor mit einem Parameter vom Typ `de.gauss.vip.config.PropertyMap` vorhanden, so wird dieser beim Start des Agenten genutzt. Der Konstruktor sollte keine zeitaufwendigen Arbeiten erledigen, da er sonst den aufrufenden Server-Thread blockiert.

Methoden

Folgende Methoden sind im Interface `ServerAgent` definiert und müssen implementiert werden.

```
package de.gauss.vip.api;

public interface ServerAgent
{
   public String getManufacturer();
   public String getDescription(Locale locale);
   public String getVersion();
   public int getRequiredVIPMajorVersion();
   public int getRequiredVIPMinorVersion();
   public boolean start(String serverType, int majorVersion,
      int minorVersion, String patchLevel);
   public void stop();
}
```

`getManufacturer`, `getDescription` und `getVersion`

Diese Methoden dienen dazu, eine Beschreibung des Agenten zurückzugeben. Die dort gemachten Angaben werden z.B. im VIP-Administrationsprogramm unter *Systemverwaltung* → *Laufende Server* → *{Servername}* → *Berichte* → *ServerAgentManager* angezeigt.

`getRequiredVIPMajorVersion` und `getRequiredVIPMinorVersion`

Diese Methoden definieren die vom Agenten benötigte Version der VIP CM Suite (so wird z.B. mit 8 für `Major` und 1 für `Minor` die VIP ContentManager-Version 8.1 spezifiziert). Ist die auf dem Server eingesetzte (d.h. laufende) Produktversion von VIP ContentManager kleiner als die vom Agenten ausgegebene Version, dann wird der Agent durch den VIP-CM-Server nicht gestartet.

`start`

Die Methode `start` wird vom VIP-CM-Server in folgenden Situationen aufgerufen:

- beim Hochfahren, genauer gesagt beim Wechsel vom Runlevel `AGENTS_STOPPED` in den Runlevel `SINGLE_USER` (siehe auch Abschnitt 4.9 “Runlevels” auf Seite 102)

  Dies hat einige Konsequenzen, da zu diesem Zeitpunkt der VIP-CM-Server noch nicht vollständig hochgefahren ist. Insbesondere kann zu diesem Zeitpunkt auch auf keine Website zugegriffen werden!

- wenn der Agent über das VIP-Administrationsprogramm explizit gestartet wird

Vor jedem Start des Agenten wird eine **neue** Instanz der Agentenklasse angelegt (unter Benutzung des Konstruktors dieser Klasse, siehe oben). Anschließend wird die `start`-Methode der Agenten-Instanz in einem eigenen Thread gestartet. Dieser Thread endet, sobald der Aufruf der Methode `start` zurückkehrt. Dies kann – abhängig von den Aktionen des Agenten – sehr kurz oder längere Zeit nach dem Aufruf erfolgen.

In der `start`-Methode kann sich der Agent auf Ereignisse registrieren, die vom VIP-CM-Server bei einer Zustandsänderung (z.B. des Systems oder einzelner VIP-Objekte) gefeuert werden. Wenn der Agent eigene Threads startet, kann auch nur der Agent selbst diese wieder beenden. In diesem Fall muss die `start`-Methode unbedingt den Wert `true` zurückgeben, damit die `stop`-Methode des Agenten bei Bedarf aufgerufen wird und diese Methode die gestarteten Threads wieder beenden kann.

Ein Beispiel für das Starten und Stoppen von Agenten finden Sie auch in Abschnitt 11.6 “Beobachten von Deployment-Ereignissen” auf Seite 257.

> **Hinweis:** Der Rückgabewert der `start`-Methode gibt an, ob der Agent erfolgreich gestartet wurde. Falls die Methode `start` den Wert `false` zurückgibt, zeigt das VIP-Administrationsprogramm den Agenten sofort als gestoppt an!

`stop`

Die Methode `stop` sollte dafür verwendet werden, belegte Ressourcen freizugeben. Sie wird aufgerufen, wenn

- die `start`-Methode den Wert `true` zurückgegeben hat (oder aufgerufen, aber noch nicht beendet wurde!)

  und entweder

- der VIP-CM-Server vom Runlevel `WEBSITE_INACCESSIBLE` in den Runlevel `AGENTS_STOPPED` wechselt

  oder

- der Agent mithilfe des VIP-Administrationsprogramms gestoppt wird

> **Hinweise**
>
> Der Rückgabewert der `start`-Methode bestimmt, ob die Methode `stop` überhaupt aufgerufen werden soll. Liefert `start` den Wert `false`, wird die `stop`-Methode niemals aufgerufen!
>
> Die `stop`-Methode sollte keine zeitaufwendigen Arbeiten erledigen, da sie sonst den aufrufenden Server-Thread blockiert.

Benachrichtigung über Ereignisse

Ein Server-Agent wird von einem Ereignisverteiler über Ereignisse informiert, auf die er sich registriert hat. Die Benachrichtigung erfolgt, solange der Agent beim Ereignisverteiler registriert ist. Damit ist die Benachrichtigung unabhängig vom Status des Agenten selbst. Sie hängt ausschließlich von der Registrierung ab.

In dem Beispiel in Abschnitt 11.6 “Beobachten von Deployment-Ereignissen” auf Seite 257 wurde die Registrierung/Deregistrierung des Agenten beim Ereignisbeobachter in die `run`-Methode des Threads verlegt. Auf diese Weise wird der Agent nur über Ereignisse informiert, solange sein eigener Thread läuft.

Beispiel

Das folgende Code-Beispiel zeigt eine Registrierung auf das Ereignis “Vollzogene Runlevel-Änderung” (`RUNLEVEL_IS`), sodass der Agent informiert wird, wenn der Server die konfigurierte Website vollständig hochgefahren hat. Weitere Informationen über die Registrierung auf Ereignisse finden Sie in Abschnitt 5.4 “Ereignisverteiler” auf Seite 128.

```
public class DemoAgent
    implements de.gauss.vip.api.ServerAgent
{
   private final static de.gauss.vip.api.event.EventDispatcher
       eventDispatcher
           = de.gauss.vip.api.VipRuntime.getEventDispatcher();

    private final de.gauss.vip.api.event.EventListener eventListener
        = new DemoRunlevelListener();

    [...]
    private final String website;
    [...]
    public boolean start(String serverType, int majorVersion,
                         int minorVersion, String patchLevel)
    {
        [...]
        eventDispatcher.addListener(null,
```

```
                              de.gauss.vip.api.event.Event.RUNLEVEL_IS,
                              eventListener);
        return true;
    }

    public void stop()
    {
        eventDispatcher.removeListener(eventListener);
        [...]
    }

    private class DemoRunlevelListener
        implements de.gauss.vip.api.event.EventListener
    {
        public void performVipEvent(de.gauss.vip.api.event.Event e)
        {
          Integer level
        =  (Integer)e.getArgument(de.gauss.vip.api.event.Event.ARG_NEW);
          String ws
     =  (String)e.getArgument(de.gauss.vip.api.event.Event.ARG_WEBSITE);

          if (level.intValue()
                    == de.gauss.vip.api.admin.Runlevel.WEBSITE_UP
                && ws != null && ws.equals(website))
                System.out.println("Website " + website + " is up");
        }
    }
}
```

3.2 Die Klasse `VipRuntime`

Die Klasse `VipRuntime` stellt den Zugriffspunkt auf die wesentlichen Schnittstellen des VIP Java API dar:

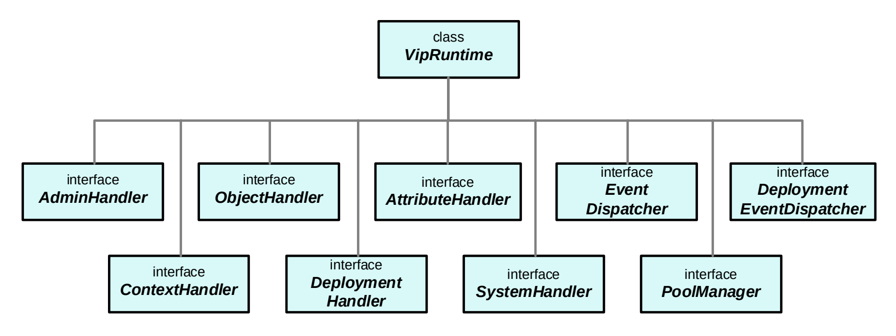


Abb. 5 – Grundlegende Interfaces des VIP Java API

Die einzelnen Interfaces bieten folgende Funktionalitäten:

- Das Interface `AdminHandler`: Diese Schnittstelle erlaubt den Zugriff auf Daten und Funktionen der Benutzer- und Systemverwaltung von VIP ContentManager. Auf diese Weise können Sie Principals anlegen und ändern, Zuordnungen bearbeiten, Informationen über Server, Websites und Deploymentsysteme abfragen sowie Runlevels auslesen und bearbeiten.

  Siehe Kapitel 4 “Administrationsinterface”.

- Das Interface `ContextHandler`: Die Schnittstelle für die Kontextverwaltung bietet Methoden an, um sich am aktuellen VIP-CM-Server anzumelden, abzumelden, das Passwort zu ändern oder das Profil eines angemeldeten Benutzers zu ermitteln.

  Siehe Kapitel 6 “Kontextverwaltung”.

- Das Interface `ObjectHandler`: Die Schnittstelle für die Objektverwaltung erlaubt den Zugriff auf die VIP-Objekte einer Website. Es werden alle aus dem VIP CMS Client (kurz: *CMS-Client*) bekannten Aktionen auf VIP-Objekte unterstützt (z.B. Anlegen, Vorlegen, Freigeben, Filtern, Löschen von VIP-Objekten).

  Siehe Abschnitt 7.1 “Das Interface ObjectHandler” ab Seite 142.

- Das Interface `AttributeHandler`: Über diese Schnittstelle ist es möglich, alle für eine Website verwendeten Attributmengen und Objektkategorien auszulesen.

  Siehe Abschnitt 7.5 “Attribute bearbeiten” ab Seite 185.

- Das Interface `DeploymentHandler`: Diese Schnittstelle ermöglicht den Zugriff auf die Metadaten von Web-Objekten wie Pfad und URL zur generierten Datei und bietet Informationen zum Status und zu Fehlern des Deployments.

  Siehe Kapitel 8 “Deployment”.

- Das Interface `SystemHandler`: Diese Schnittstelle ermöglicht den Zugriff auf Funktionen der Systemverwaltung. Dazu gehören das Lesen von Server-Berichten und -Protokollen sowie das Tracing.

  Siehe Kapitel 10 “Systemverwaltung”.

- Die Interfaces `EventDispatcher` und `DeploymentEventDispatcher`: Die Ereignisverwaltung bietet eine Schnittstelle für die Registrierung (und Deregistrierung) von Ereignisbeobachtern. Es handelt sich dabei um eine für den VIP-CM-Server zentrale Instanz, die Ereignisse an registrierte Beobachter verteilt. Diese Ereignisse können sich auf das gesamte VIP-CM-System, Websites oder Deploymentsysteme beziehen.

Agenten können sich auf Ereignisse vor dem Ausführen einer Aktion (`PrepareEvent`) oder erst nach erfolgreicher Ausführung (`Event`) registrieren. Im ersten Fall kann eine Aktion durch das Einlegen eines Vetos (in Form einer `Exception`) abgebrochen werden.

Siehe Kapitel 5 “Ereignisverarbeitung”.

- Das Interface `PoolManager`: Diese Schnittstelle ermöglicht den Zugriff auf Verbindungen, die auf eigenen Pooltypen (z.B. für die Anbindung von Fremdsystemen) basieren. Diese Verbindungsarten können über den Pool-Mechanismus von VIP ContentManager verwaltet werden.

  Siehe Kapitel 9 “Poolverwaltung”.

```
public final class VipRuntime
{
    static public ObjectHandler getObjectHandler (String websiteName);
    static public ObjectHandler getObjectHandler (String websiteName,
                                                  String serverType);
    static public AttributeHandler getAttributeHandler
       (String websiteName);
    static public AdminHandler getAdminHandler ();
    static public ContextHandler getContextHandler ();
    static public EventDispatcher getEventDispatcher ();
    static public DeploymentEventDispatcher
                  getDeploymentEventDispatcher ();
    static public DeploymentHandler
                  getDeploymentHandler(String website);
    static public Server getCurrentServer ();
    static public PoolManager getPoolManager();
    static public SystemHandler getSystemHandler();
}
```

3.3 Fehlerbehandlung

Das VIP Java API signalisiert Fehler in VIP ContentManager durch den Exception-Mechanismus von Java. Jede Methode, die einen Fehler liefern kann, wirft im VIP Java API eine Exception. Zu den Exception-Klassen können lokalisierte Fehlermeldungen ausgegeben werden (siehe Abschnitt “Lokalisierte Fehlermeldungen” auf Seite 51).

Basisklasse `VipApiException`

Neben der Basisklasse `VipApiException` bietet das VIP Java API eine Reihe von Exception-Klassen, die von `VipApiException` ableiten. Diese Exceptions können aufgrund von Fehlersituationen während der Ausführung einer Methode geworfen werden.

Lokalisierte Fehlermeldungen

Die Methoden `getMessage` bzw. `getMessages` können dazu verwendet werden, lokalisierte Texte zu einer Exception zurückzugeben.

`getMessage(java.util.Locale locale)`

Die Methode liefert einen lokalisierten Text, der die Ursache des Fehlers beschreibt.

`getMessages(java.util.Locale locale)`

Die Methode liefert alle lokalisierten Meldungen des Fehlers als String-Array zurück.

Beispiel

Das fehlgeschlagene Vorlegen eines Objekts könnte über `getMessages` folgende Meldungen zurückgeben:

- Erste Meldung: “Das Objekt 4711 konnte nicht vorgelegt werden.”
- Zweite Meldung: “Das übergeordnete Thema wurde noch nicht vorgelegt.”

In diesem Fall ist es also die zweite Meldung, aus der die eigentliche Ursache des Fehlers hervorgeht. Die erste Meldung beschreibt das Resultat, nämlich dass das Objekt nicht vorgelegt werden konnte.

```
public class VipApiException extends Exception
{
   public VipApiException()

   public String getMessage()
   public String getMessage(Locale locale)
   public String[] getMessages(Locale locale)
   public Exception getWrappedException()
   public Object[] getArguments()
   public void setWrappedException(Exception ex, Object[] args)
   public String getLocalizedMessage()
   public String toString()
}
```

Abgeleitete Exception-Klassen

In den folgenden Abschnitten werden die von `VipApiException` abgeleiteten Exception-Klassen kurz umrissen. Genauere Angaben hierzu finden Sie in der Online-Dokumentation (Javadoc).

`AccessDeniedException`

Die Aktion ist aufgrund der für das Objekt definierten Zugriffsrechte nicht erlaubt.

Mögliche Ursache

- In der Zugriffssteuerungsliste des Objekts hat der aktuelle Benutzer nicht die erforderlichen Rechte, um die jeweilige Aktion auszuführen.

`ActionNotPermittedException`

Die Aktion ist nicht zulässig.

Mögliche Ursachen

- Es wurden unerlaubte Metadaten verwendet.
- Der Status der Vorlage bzw. des übergeordneten Navigationsobjekts ist nicht richtig.
- Die Aktion wurde an einem Server ausgeführt, an dem sie nicht erlaubt ist (z.B. `create` auf QS-Server).

`AttributeException`

Die verwendeten Attribute bzw. Attributwerte sind nicht korrekt oder die versuchte Änderung der Attribute ist fehlgeschlagen.

Mögliche Ursachen

- Die Metadaten in einem `RepositoryEntry` enthalten unbekannte Attribute bzw. unerlaubte Attributwerte.

- Es wurde versucht, Attribute zu ändern, die nicht änderbar sind.
- Es wurden Attribute für die Suche verwendet, die nicht suchbar sind.

> **Hinweis:** Informationen zu änderbaren und suchbaren Attributen enthält die Tabelle 13 “Standardmetadaten” auf Seite 188.

`DatabaseException`

Eine Datenbankoperation ist fehlgeschlagen.

Mögliche Ursachen

- Die Datenbank ist nicht verfügbar.
- Der VIP-CM-Server ist mit einer Kennung an der Datenbank angemeldet, die keine ausreichenden Zugriffsrechte beinhaltet.

`DeploymentSystemNotFoundException`

Es wurde eine Methode aufgerufen, bei der der Name eines Deploymentsystems angegeben werden muss. Das angegebene Deploymentsystem konnte jedoch nicht gefunden werden.

Mögliche Ursachen

- Der Name des Deploymentsystems ist unbekannt oder wurde falsch angegeben.
- Das angegebene Deploymentsystem existiert in der benutzten Website nicht.

`FileException`

Eine Dateioperation ist fehlgeschlagen.

Mögliche Ursache

- Der VIP-CM-Server ist mit einer Kennung (am Betriebssystem bzw. im Netzwerk) angemeldet, die keine ausreichenden Zugriffsrechte im Dateisystem beinhaltet.

`InvalidContextIdException`

Der Kontext ist ungültig.

Mögliche Ursachen

- Nach einer erfolgreichen Anmeldung wurde längere Zeit keine Aktion mehr durchgeführt, sodass der Kontext ungültig wurde.
- Statt des erwarteten Benutzerkontextes wurde ein Transaktionskontext verwendet.

`InvalidObjectException`

Die versuchte Änderung an den Metadaten eines Objekts kann nicht durchgeführt werden, weil die Objektdaten nicht (mehr) gültig sind.

Mögliche Ursache

- Die Metadaten des Objekts wurden von einem anderen Benutzer geändert. Die geänderten Metadaten wurden noch nicht gelesen.

`LicenseException`

Die erforderlichen Lizenzbedingungen sind nicht erfüllt.

Mögliche Ursache

- Die Lizenz ist nicht mehr gültig.

`LoginException`

Die Anmeldung ist fehlgeschlagen.

Mögliche Ursachen

- Die Benutzerkennung ist unbekannt oder falsch.
- Das Kennwort ist falsch.

`MailException`

Das Versenden von E-Mails als Ergebnis einer Aktion ist fehlgeschlagen.

Mögliche Ursachen

- Der Mailserver ist nicht erreichbar.
- Die Empfängerangaben sind fehlerhaft.

`NetException`

Es ist ein Kommunikationsfehler über die Socket-Verbindung (VIPP-Port) aufgetreten.

Mögliche Ursachen

- Die Verbindung ist unterbrochen.
- Es steht keine Verbindung mehr aus dem Pool zur Verfügung.

`ObjectInUseException`

Die Aktion an einem Objekt konnte nicht ausgeführt werden, weil ein anderer Benutzer das Objekt gerade im (exklusiven) Zugriff hat.

Mögliche Ursachen

- Das Objekt ist gerade durch eine andere Aktion (z.B. Import, Verschieben) gesperrt.
- Das Objekt wird gerade durch einen parallel arbeitenden Benutzer bzw. Agenten geändert.

`ObjectNotFoundException`

Das gesuchte Objekt wurde im Web-Content-Repository nicht gefunden.

Mögliche Ursachen

- Der angemeldete Benutzer besitzt nicht das Leserecht für das Objekt.
- Die angegebene OID existiert nicht in der angegebenen Website.
- Das Objekt wurde zerstört (endgültig gelöscht).

`RunlevelException`

Die Methode konnte nicht aufgerufen werden, weil der VIP-CM-Server sich in einem Runlevel befindet, der diesen Zugriff nicht erlaubt.

Mögliche Ursachen

- Der VIP-CM-Server wird gerade heruntergefahren.
- Der Runlevel wurde heruntergesetzt (z.B. um eine Datensicherung durchzuführen).

`SearchException`

Die Suche nach einem VIP-Objekt über eine Filteroperation ist fehlgeschlagen.

Mögliche Ursache

- Die Filterparameter wurden nicht korrekt angegeben, sodass keine SQL-Anweisung für die Suche in der Datenbank erstellt werden konnte.

`UnexpectedException`

Die Methode hat eine Exception gefangen, die nicht zu den für die Klasse bzw. das Interface definierten Exceptions gehört.

`VetoException`

Gegen die Methode wurde ein Veto eingelegt.

Mögliche Ursache

- Ein Server-Agent hat sich für das entsprechende Vorbereitungsereignis registriert und gegen die versuchte Änderung ein Veto eingelegt.

`WebsiteNotFoundException`

Die gesuchte Website konnte nicht gefunden werden.

Mögliche Ursache

- Die angegebene Website existiert nicht im VIP-CM-System.

3.4 Lokalisierte Meldungen

Für die Repräsentation von lokalisierten Meldungen gibt es im VIP Java API die Basisklasse `UserMessage` (`de.gauss.vip.api.UserMessage`). Die wichtigste Methode dieser Klasse ist `getString(locale)`. Diese liefert eine Repräsentation der Meldung in der angegebenen Sprache (ohne Berücksichtigung des Ländercodes).

Eine Meldung besteht dabei immer aus einem Schlüssel (Key) in Form einer sprachunabhängigen Zeichenkette und optional mehreren Argumenten (beliebige Java-Objekte). Die Übersetzung des sprachunabhängigen Schlüssels wird aus einer Message-Datei gelesen. Für jede Message-Klasse gibt es pro unterstützte Sprache eine Datei, in der die Zuordnungen von Schlüsseln (Message-Keys) zu Meldungstexten enthalten sind.

Die Message-Dateien befinden sich im VIP-Installationsverzeichnis unter **\config\resource**, **\admin\config\resource** sowie **\contentminer\config\resource**. Die Dateinamen sind wie folgt aufgebaut:

<className>_<langCode>.properties.

Die Sprachkennung (`<langCode>`) im Dateinamen der Message-Dateien basiert auf ISO 639 (eine Liste der Sprachkennungen nach ISO 639 ist unter http://www.w3.org/WAI/ER/IG/ert/iso639.htm zu finden). Für benutzerdefinierte Meldungen in Deutsch ergibt sich die Datei

UserMessage_de.properties.

Damit die lokalisierten Meldungen auf den verschiedenen Servern des VIP-CM-Systems verfügbar sind, müssen die entsprechenden Konfigurationsdateien in die dafür vorgesehenen Verzeichnisse der Server kopiert werden.

Beispiel

Zur Verdeutlichung nehmen wir einen einfachen Begrüßungstext, der im Code mit dem folgenden (sprachunabhängigen) Schlüssel angegeben ist:

“USER_WELCOME”

Diese Meldung besitzt zwei Argumente, “{0}” und “{1}”. Für beide Argumente wird jeweils die Methode toString aufgerufen.

In der Properties-Datei **UserMessage_en.properties** wird der englische Meldungstext für diesen Schlüssel wie folgt angegeben.

```
...
USER_WELCOME=Hello, {0}. Welcome to ''{1}''.
...
```

Wenn der aktuelle Benutzer die Kennung “jstein” hat und in der Website “InternetSite” angemeldet ist, ergibt der folgende Code die Meldung: “Hello, jstein. Welcome to ‘InternetSite’.”

```
UserMessage msg = new UserMessage("USER_WELCOME",
   userName, webSiteName);
System.out.println(msg.getString(Locale.en_US));
```

KAPITEL 4

Administrationsinterface

Das Administrationsinterface des VIP Java API – der `AdminHandler` – erlaubt den Zugriff auf Funktionen der Benutzer- und der Systemverwaltung von VIP ContentManager.

Funktionen der Benutzerverwaltung

- Principals anlegen und löschen, Profile von Principals ermitteln und bearbeiten, Principals suchen (siehe Abschnitt 4.1 “Principals bearbeiten” auf Seite 67)
- LDAP-Principals mit VIP-Attributen ausstatten und ihnen damit Zugriff auf das VIP-CM-System ermöglichen (siehe Abschnitt 4.2 “VIP-Attribute in LDAP setzen” auf Seite 79)
- Principals suchen (siehe Abschnitt “Principals suchen” ab Seite 73)
- Rechte von Principals bearbeiten (siehe Abschnitt 4.3 “Rechte von Principals bearbeiten” auf Seite 82)
- Funktionsbereiche anlegen und löschen sowie Informationen über Funktionsbereiche ermitteln (siehe Abschnitt 4.4 “Funktionsbereiche bearbeiten” auf Seite 89)
- Zuordnen von Benutzern zu Gruppen bzw. Rollen sowie von Funktionsbereichen und Websites zu Principals; Aufheben dieser Zuordnungen (siehe Abschnitt 4.5 “Zuordnungen von Principals bearbeiten” auf Seite 94)

Funktionen der Systemverwaltung

- Ermittlung von Informationen über die verwendeten VIP-CM-Server (siehe Abschnitt 4.6 “Informationen zu VIP-CM-Servern” auf Seite 96)
- Ermittlung von Informationen über Websites (siehe Abschnitt 4.7 “Informationen zu Websites” auf Seite 99)
- Ermittlung von Informationen über Deploymentsysteme (siehe Abschnitt 4.8 “Informationen zu Deploymentsystemen” auf Seite 101)
- Auslesen und gegebenenfalls Ändern des Systemzustands (Runlevel) von VIP-CM-Servern und Websites (siehe Abschnitt 4.9 “Runlevels” auf Seite 102)

Eine Instanz des Administrationsinterfaces erhalten Sie über `VipRuntime.getAdminHandler`.

> **Hinweis:** Der Zugriff auf die Daten der Benutzer- und Systemverwaltung setzt entsprechende Administrationsrechte voraus, z.B. das Recht “Principal anlegen, ändern, löschen” für das Anlegen und Bearbeiten von Principals sowie “Zugriff auf Systemverwaltung” für das Lesen von Systeminformationen. Einen Überblick über die Administrationsrechte bietet Tabelle 2 “Die einzelnen Administrationsrechte” auf Seite 30.

Eine kurze Beschreibung der einzelnen Methoden des Administrationsinterfaces wird in den folgenden Abschnitten gegeben. Soweit nicht anders angegeben, befinden sich alle Klassen und Interfaces im Package `de.gauss.vip.api.admin`.

```
package de.gauss.vip.api.admin
public interface AdminHandler
{
  // --- Constants to determine directly or indirectly assigned
  //     users, websites, and functional areas.
  public static int NO_ASSIGNMENTS;
  public static int DIRECT_ASSIGNMENTS;
  public static int INDIRECT_ASSIGNMENTS;

  // --- Constants to define the type of a principal. These constants
  //     are used in retrieval methods, e.g. getPrincipalsForWebsite to
  //     specify the type of principals to be searched
  public static final int GROUP_PRINCIPALS;
  public static final int ROLE_PRINCIPALS;
  public static final int USER_PRINCIPALS;
  public static final int ALL_PRINCIPALS;

  //--- Factories for the creation of principals and functional areas
  public PrincipalFactory getPrincipalFactory(ContextId cid);
  public FunctionalAreaFactory getFunctionalAreaFactory(ContextId cid);

  //--- add, remove and update principals and functional areas
  public void add(ContextId id, Principal principal);
  public void add(ContextId id, FunctionalArea funcarea);
  public void remove(ContextId id, Principal principal);
  public void remove(ContextId id, FunctionalArea funcarea);
  public void update(ContextId id, Principal principal);

  //--- retrieve LDAP users and set VIP attributes in LDAP thus enabling
  //    LDAP users to access the VIP CM system
  public List getUnassignedLDAPPrincipals(ContextId id, String
                                          ldapContext, boolean recursive);
  public void importLDAPPrincipalAsUser(ContextId id, StringValue
                                          ldapdn);
  public void importLDAPPrincipalAsGroup(ContextId id, StringValue
                                          ldapdn);
  public void importLDAPPrincipalAsRole(ContextId id, StringValue
                                          ldapdn );

  //--- create and remove assignments between principals, websites, and
  //    functional areas
  public void defineAssignment(ContextId id, User arg1, Principal arg2);
  public void defineAssignment(ContextId id, FunctionalArea fa,
                               Principal principal);
  public void defineAssignment(ContextId id, Website ws, Principal
                               principal);
  public void removeAssignment(ContextId id, User arg1, Principal arg2);
```

```
public void removeAssignment(ContextId id, FunctionalArea fa,
                             Principal principal);
public void removeAssignment(ContextId id, Website ws, Principal
                             principal);

//--- editing rights of principals
public void setAdminPermissions(ContextId id, Principal principal,
                                List granted );
public void setInitialRights(ContextId id, Principal principal,
                             InitialPrincipalRights perm );

// --- Reading profiles of principals
public User getUserProfile(ContextId cid, String userName,
                            int assignmentFlag);
public Group getGroupProfile(ContextId cid, String groupName);
public Role getRoleProfile(ContextId cid, String roleName);

//--- retrieve principals
public List getPrincipals(ContextId id, int princType, String
                          ldapContext, Filter expression, int startPos,
                          int maxResults) throws VipApiException;
public List getPrincipalsForWebsite(ContextId id, int princType,
                          int maxResults, String website,
                          String namePrefix);
public List getUsersForWebsite(ContextId id, int maxResults,
                               String website, String namePrefix);
public List getGroupsForWebsite(ContextId id, int maxResults,
                               String website,String namePrefix);
public List getRolesForWebsite(ContextId id, int maxResults,
                               String website, String namePrefix );

// --- Information on functional areas
public FunctionalArea getFunctionalArea(ContextId cid,
    String functionalAreaName, int assignmentFlag);
public List getFunctionalAreas(ContextId cid);

// --- Generating users, groups and roles without profile
//     For use in ACLs as a principal
public User getUser(String userName);
public Group getGroup(String groupName);
public Role getRole(String roleName);

// --- Retrieve servers
public Server getServer(ContextId cid, String serverName);
public List getServers(ContextId cid);

// --- Retrieve websites
public Website getWebsite(ContextId cid, String websiteName,
                          int assignmentFlag);
```

```
  public List getWebsites(ContextId cid);

  // --- Retrieve deployment systems
  public DeploymentSystem getDeploymentSystem(ContextId cid,
                                              String dplName);
  public List getDeploymentSystems(ContextId cid);

  // --- Reading and changing the runlevels
  public int getRunlevel(ContextId cid, String serverName,
                         String websiteName);
  public void setRunlevel(ContextId cid, String serverName,
                          String websiteName, int level);
}
```

4.1 Principals bearbeiten

Das Administrationsinterface stellt Methoden zum Bearbeiten von Benutzern, Gruppen und Rollen zur Verfügung.

In diesem Abschnitt werden die Methoden für folgende Zielstellungen erläutert:

- Principals anlegen (siehe “Principals anlegen” auf Seite 68)
- Einstellungen von Principals bearbeiten (siehe “Profile von Principals bearbeiten” auf Seite 70)
- Principals suchen (siehe “Principals suchen” auf Seite 73)
- Principals löschen (siehe “Principals löschen” auf Seite 78)

Hinweis: Das Bearbeiten und Löschen von Principals kann unabhängig davon erfolgen, ob die Benutzerdaten in einem RDBMS oder einem LDAP-Verzeichnisdienst gespeichert sind. Die Möglichkeit, Principals anzulegen, ist vom verwendeten LDAP-Verzeichnisdienst abhängig (siehe Installationshandbuch zur VIP CM Suite).

Principals anlegen

Das Anlegen von Principals erfolgt über die `PrincipalFactory`. Die Factory dient lediglich zum Erstellen der Principal-Objekte. Um die neu erstellten Benutzer, Gruppen oder Rollen dauerhaft dem VIP-CM-System hinzuzufügen, muss anschließend die `add`-Methode des `AdminHandler`-Interfaces aufgerufen werden.

Das Interface `PrincipalFactory` ist folgendermaßen aufgebaut:

```
public interface PrincipalFactory
{
  public User createUser(String userID, String cn, String ldapPosition,
                         String email, LocaleValue language,
                         String vipUserpassword, boolean initPassword);
  public Role createRole(String cn, String ldapPosition, String email);
  public Group createGroup(String cn, String ldapPosition,
                           String email);
}
```

Eine Instanz der `PrincipalFactory` kann über die Methode `getPrincipalFactory` des `AdminHandler`-Interfaces angefordert werden.

Beim Anlegen eines Principals über eine der Methoden `createUser`, `createRole` oder `createGroup` werden verschiedene Parameter mit den Eigenschaften des neuen Principals übergeben.

Benutzer, Gruppen und Rollen verfügen über eine Reihe von gemeinsamen Eigenschaften, die beim Anlegen gesetzt werden können:

- einen Namen (`cn`)
- eine LDAP-Position (`ldapPosition`) – Nur bei Verwendung eines LDAP-Verzeichnisdienstes zur Speicherung der Benutzerdaten. Bei Verwendung eines RDBMS ist `null` zu übergeben.
- eine E-Mail-Adresse (`email`)

Darüber hinaus müssen für Benutzer weitere Eigenschaften gesetzt werden:

- die Benutzerkennung zur Anmeldung am VIP-CM-System (`userID`)
- das Passwort für die Anmeldung (`vipUserpassword`)
- ein Flag, ob bei der ersten Anmeldung am VIP-CM-System das Passwort geändert werden muss (`initPassword`)
- die bevorzugte Sprache des Benutzers (`language`)

> **Hinweis:** Die Erlaubnis zum Zugriff auf das VIP-CM-System (`vipAccess`) wird beim Anlegen von Principals implizit gesetzt und aktiviert.

Das folgende Beispiel legt einen neuen LDAP-Benutzer an:

```
// login with appropriate administration rights
ContextId cid = VipRuntime.getContextHandler().login(...);

// define the LDAP position of the new user
String ldapRootPos = "uid=admin, o=test, c=de";

// define the name (cn) and ID of the new user
String userCN = "Joshua Stein";
String userId = "jstein";

// define passwort, e-mail address and language of the new user
String passwd = "4711";
String email = "joshua.stein@company.com";
LocaleValue loc = new LocaleValue("en_US");

// create the new user by means of the PrincipalFactory
AdminHandler aH = VipRuntime.getAdminHandler();
PrincipalFactory princFac = aH.getPrincipalFactory(cid);
User usrNew = princFac.createUser(userId, userCN, ldapRootPos,
              email, loc, passwd, true);

// add the new user to the VIP CM system by means of the AdminHandler
// method void add(ContextId, Principal)
aH.add( cid, usrNew );
```

Profile von Principals bearbeiten

Jeder Principal verfügt über eine Menge von Eigenschaften, die in ihrer Gesamtheit das Profil des Principals bilden. Benutzer, Gruppen und Rollen besitzen eine Reihe von gemeinsamen Eigenschaften wie Name und E-Mail-Adresse. Darüber hinaus können bei Gruppen und Rollen zugehörige Benutzer angegeben werden. Benutzer verfügen zusätzlich über Eigenschaften wie Name und Passwort für die Anmeldung am VIP-CM-System, eine Sprache und einen Stellvertreter.

Hinweise:

Nicht alle Eigenschaften eines Principals können nachträglich geändert werden, z.B. die Benutzerkennung oder der Name einer Gruppe bzw. Rolle.

Informationen zum Ändern der Administrations- und Standard-Objektrechte von Principals erhalten Sie im Abschnitt 4.3 “Rechte von Principals bearbeiten” auf Seite 82. Hinweise zum Bearbeiten der Zuordnungen von Principals enthält Abschnitt 4.5 “Zuordnungen von Principals bearbeiten” auf Seite 94.

Die Profile von Principals können über die Funktionen des VIP Java API gelesen und bearbeitet werden. Dazu dienen die Methoden der Interfaces `User`, `Group` und `Role`. Diese Interfaces sind folgendermaßen definiert:

```
public interface User extends Principal, Value
{
  public boolean hasProfile();
  public boolean isPasswordChangeRequired();
  public String getUserId();
  public String getCommonName();
  public String getEMailAddress();
  public boolean getVipAccess();
  public InitialPrincipalRights getInitialRights();
  public Locale getLocale();
  public User getSubstitute();
  public List getSubstituteOf();
  public List getGroups();
  public List getRoles();
```

```
  public List getFunctionalAreas();
  public List getWebsites();
  public Boolean getTrustedLogin();
  public String getVIPDN();
  public Set getDomains();
  public List getAdminPermissions();
  public void setCommonName(String cn);
  public void setEMailAddress(String email);
  public void setPassword(String pwd);
  public void setVipAccess(boolean access);
  public void setInitPasswordFlag(boolean initPwd);
  public void setLocale(LocaleValue locale);
  public void setTrustedLogin(boolean trustedLogin);
  public void setDomains(Set domains);
}

public interface Role extends Principal, Value
{
  public boolean hasProfile();
  public String getCommonName();
  public String getEMailAddress();
  public boolean getVipAccess();
  public InitialPrincipalRights getInitialRights();
  public List getFunctionalAreas();
  public List getWebsites();
  public List getMembers();
  public void setEMailAddress(String mail);
  public void setVipAccess(boolean access);
  public String getVIPDN();
  public List getAdminPermissions();
}

public interface Group extends Principal, Value
{
  public boolean hasProfile();
  public String getCommonName();
  public String getEMailAddress();
  public boolean getVipAccess();
  public InitialPrincipalRights getInitialRights();
  public List getFunctionalAreas();
  public List getWebsites();
  public List getMembers();
  public void setEMailAddress(String mail);
  public void setVipAccess(boolean access);
  public String getVIPDN();
  public List getAdminPermissions();
}
```

Profil ermitteln

Das Profil eines Benutzers, einer Gruppe oder einer Rolle wird mithilfe der Methode `getUserProfile` des `AdminHandler`-Interfaces ermittelt.

Das folgende Beispiel ermittelt das Profil des Benutzers mit der Kennung “jstein”. Das Profil enthält in diesem Beispiel auch alle indirekt zugeordneten Funktionsbereiche und Websites, d.h. die Funktionsbereiche und Websites der Rollen und Gruppen von jstein.

```
AdminHandler ah = VipRuntime.getAdminHandler();
ContextId cid = VipRuntime.getContextHandler().login("admin", password);
User user = ah.getUserProfile(cid, "jstein",
            AdminHandler.INDIRECT_ASSIGNMENTS);
List websites = user.getWebsites();
Iterator i = websites.iterator();

while (i.hasNext())
{
   Website w = (Website)i.next();
   System.out.println("User '"+user.getName()+
                      "' is assigned to website '"+w.getName()+"'.");
}
```

Profile bearbeiten

Zur Bearbeitung des Profils wird zunächst das Profil des entsprechenden Benutzers geladen. Mithilfe der entsprechenden Methoden aus dem Interface `User`, `Group` oder `Role` werden die gewünschten Eigenschaften gesetzt. Um die Änderungen dauerhaft in der Benutzerverwaltung zu speichern, wird abschließend die Methode `update` des `AdminHandler`-Interfaces aufgerufen.

Das folgende Beispiel ändert die E-Mail-Adresse und die Spracheinstellung des Benutzers mit der Kennung “jstein”.

```
// login
ContextId cid = VipRuntime.getContextHandler().login(...);

// load user profile of "jstein"
AdminHandler aH = VipRuntime.getAdminHandler();
User usrJOS = aH.getUserProfile(cid, "jstein",
              AdminHandler.INDIRECT_ASSIGNMENTS);

//set new e-mail address and locale for the user
usrJOS.setLocale(new LocaleValue("de_DE") );
usrJOS.setEMailAddress(jstein@company.com);

//save changes in the user administration
aH.update(cid, usrJOS);
```

Principals suchen

Mithilfe der Methode `getPrincipals` des `AdminHandler`-Interfaces kann nach Principals gesucht werden, die ein bestimmtes Kriterium erfüllen. Das Suchkriterium wird durch ein Objekt vom Typ `Filter` repräsentiert.

> **Hinweis:** Ausführliche Informationen zur Klasse `Filter` erhalten Sie im Abschnitt 7.6 “VIP-Objekte suchen” ab Seite 195.

Die gefundenen Principals besitzen jedoch kein Profil, dieses muss anschließend mithilfe der Methoden `getUserProfile`, `getGroupProfile` bzw. `getRoleProfile` geladen werden.

Das Interface `SearchableKeys` stellt Konstanten zur Verfügung, die als Suchkriterien verwendet werden können:

USER_ID

Mit dieser Konstante kann nach Benutzern mit einer bestimmten Kennung gesucht werden, z.B. über einen `EqualFilter` oder `LikeFilter`.

Klasse des Werts

`StringValue`

Beispiel

Der folgende Filter sucht nach dem Benutzer mit der Kennung “jstein”:

```
new LikeFilter( SearchableKeys.USER_ID, new StringValue("jstein*") );
```

VIP_DN

Mit dieser Konstante kann nach LDAP-Principals mit einem bestimmten “distinguished name” gesucht werden.

Klasse des Werts

`StringValue`

Beispiel

Der folgende Filter sucht nach dem Principal mit dem distinguished name “jstein”:

```
new EqualFilter( SearchableKeys.VIP_DN,
                new StringValue("user_id=jstein") );
```

COMMON_NAME

Mit dieser Konstante kann nach Principals mit einem bestimmten Namen gesucht werden.

Klasse des Werts

`StringValue`

Beispiel

Der folgende Filter sucht nach Principals mit dem Namen “Joshua Stein”:

```
new LikeFilter( SearchableKeys.COMMON_NAME, new StringValue("Josua
                                                         Stein*") );
```

LANGUAGE

Mit dieser Konstante kann nach Benutzern mit einer bestimmten Spracheinstellung gesucht werden.

Klasse des Werts

`LocaleValue`

Beispiel

Der folgende Filter sucht nach Benutzern mit der Spracheinstellung “Englisch (USA)”:

```
new EqualFilter( SearchableKeys.LANGUAGE, new LocaleValue("en_US" ) );
```

TRUSTED_LOGIN

Mit dieser Konstante kann nach Benutzern gesucht werden, bei denen die Einstellung “vertraute Anmeldung” gesetzt bzw. nicht gesetzt ist.

Klasse des Werts

BooleanValue

Beispiel

Der folgende Filter sucht nach Benutzern, denen eine vertraute Anmeldung gestattet ist:

```
new EqualFilter( SearchableKeys.TRUSTED_LOGIN,
                new BooleanValue( true ));
```

VIP_ACCESS

Mit dieser Konstante kann nach Principals gesucht werden, in deren Einstellungen der Zugriff auf das VIP-CM-System aktiviert bzw. deaktiviert ist.

Klasse des Werts

BooleanValue

Beispiel

Der folgende Filter sucht nach Principals, deren Zugriff auf das VIP-CM-System deaktiviert wurde:

```
new EqualFilter( SearchableKeys.VIP_ACCESS,
                 new BooleanValue( false ));
```

MAIL

Mit dieser Konstante kann nach Principals mit einer bestimmten E-Mail-Adresse gesucht werden.

Klasse des Werts

`StringValue`

Beispiel

Siehe folgenden Abschnitt

Beispiele

Das folgende Beispiel sucht alle Principals, in deren E-Mail-Adresse der String “company.com” enthalten ist. Die Suche wird im LDAP-Knoten “o=company, c=de” durchgeführt.

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//construct filter expression for searching principals with the domain
//"company.com" in their e-mail address. A LikeFilter is used.
LikeFilter mailFilter = new LikeFilter(SearchableKeys.MAIL,
                                       new StringValue("*company.com"));

//search principals based on this filter
List principals = ah.getPrincipals(cid, AdminHandler.USER_PRINCIPALS,
"o=company, c=de", mailFilter, 0, -1);
```

Das zweite Beispiel zeigt, wie zwei Filter miteinander kombiniert werden. Es wird nach allen Principals gesucht, die die Spracheinstellung Englisch haben und bei denen der Zugriff auf das VIP-CM-System deaktiviert ist.

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//construct filter expression for searching principals with
```

```
//deactivated access to the VIP CM system
EqualFilter vipAccessFilter = new EqualFilter(SearchableKeys.VIP_ACCESS,
                                              new BooleanValue(false));
//construct filter expression for searching principals with
//language setting=English
//combine this filter with the vipAccessFilter constructed above
EqualFilter languageFilter = new EqualFilter(SearchableKeys.LANGUAGE,
                                             new LocaleValue("en_US"));
AndFilter filterCombination = new AndFilter(vipAccessFilter,
                                            languageFilter);
//perform searching, the principals found have the language setting
//"English" and their access to the VIP CM system is deactivated
List principals = ah.getPrincipals(cid, AdminHandler.USER_PRINCIPALS,
"o=company, c=de", filterCombination, 0, -1);
```

Principals löschen

Das Löschen von Principals erfolgt mithilfe der Methode `remove` des `AdminHandler`-Interfaces. Der entsprechende Principal wird vollständig aus dem VIP-CM-System entfernt und aus allen entsprechenden Zuordnungen gelöscht.

> **Hinweis:** Wenn Sie einen LDAP-Principal im VIP-CM-System löschen, wird dieser vollständig aus dem LDAP-Verzeichnisdienst entfernt.

4.2 VIP-Attribute in LDAP setzen

Wird für die Verwaltung der Benutzerdaten des VIP-CM-Systems ein LDAP-Verzeichnisdienst verwendet, müssen die im LDAP-Server gespeicherten Principals mit den VIP-Attributen ausgestattet werden. Auf diese Weise wird ihnen der Zugriff auf das VIP-CM-System ermöglicht.

> **Hinweis:** Informationen zur Konfiguration der LDAP-Anbindung für die VIP CM Suite erhalten Sie im Installationshandbuch.

Das Setzen der erforderlichen VIP-Attribute für Principals im LDAP-Verzeichnisdienst kann auch über das VIP Java API erfolgen.

Mithilfe der Methode `getUnassignedLDAPPrincipals` können die Principals innerhalb eines bestimmten Suchknotens abgefragt werden, die noch nicht mit den VIP-Attributen ausgestattet sind. Als Ergebnis wird eine Liste zurückgeliefert, die die “distinguished names” (DN) der ermittelten Principals als `StringValues` enthält.

Mit den Methoden `importLDAPPrincipalAsUser`, `importLDAPPrincipalAsGroup` und `importLDAPPrincipalAsRole` können diese Principals dann mit den VIP-Attributen ausgestattet werden. Als Parameter ist neben der `ContextId` jeweils ein `StringValue` mit dem DN des zu importierenden Principals anzugeben.

Beispiel

Im LDAP-Verzeichnisdienst gibt es einen Knoten “o=company, c=de”. Unterhalb dieses Knotens gibt es u.a. einen Knoten für Benutzer (“ou=users”) und einen Knoten für Gruppen (“ou=groups”). In diesen beiden Knoten befinden sich die Principaleinträge, die noch nicht über die VIP-Attribute verfügen und über das VIP Java API mit diesen Attributen ausgestattet werden sollen.

Die folgende Grafik veranschaulicht die LDAP-Struktur:

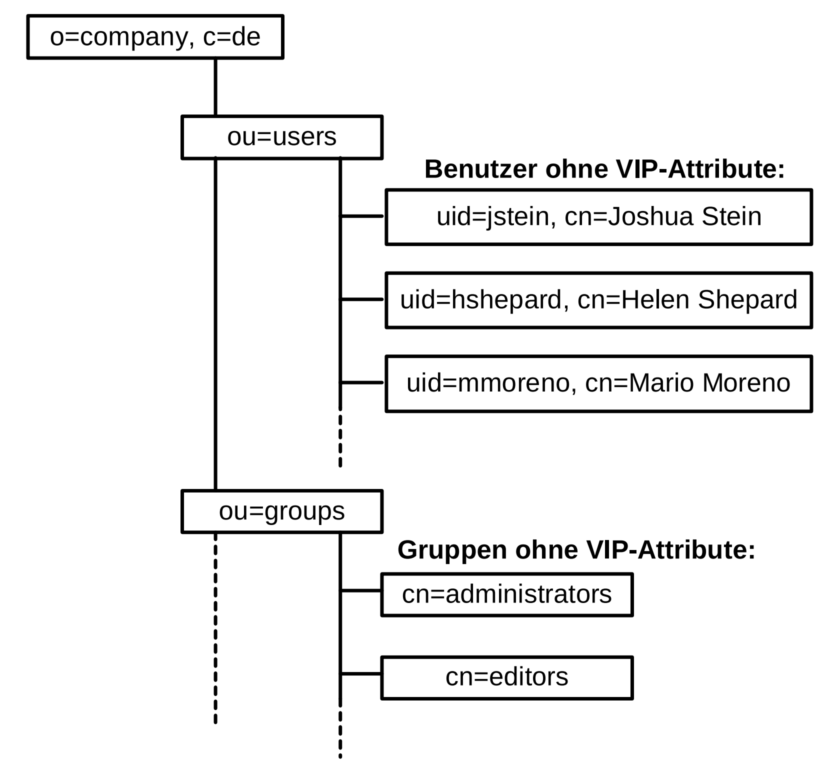


Abb. 6 – Beispielhaftes LDAP-Schema

Das folgende Beispiel sucht in den Knoten ou=users und ou=groups nach Principals, die noch nicht über die VIP-Attribute verfügen, und stattet die gefundenen Principals mit den Attributen aus.

```
// login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//retrieve principals without VIP attributes in the LDAP context
//"o=company, c=de, ou=users"
//The returned list contains the LDAP DNs as StringValue objects
List unassignedUsers = ah.getUnassignedLDAPPrincipals(cid, "o=company,
                                                c=de, ou=users", false);

//set VIP attributes for these users
Iterator usrIter = unassignedUsers.iterator();
while ( usrIter.hasNext() )
{
  StringValue usrDN = (StringValue) usrIter.next();
  ah.importLDAPPrincipalAsUser(cid, usrDN);
}

//retrieve principals without VIP attributes in the LDAP context
//"o=company, c=de, ou=groups" and assign the VIP attributes to
//these principals
List unassignedGroups = ah.getUnassignedLDAPPrincipals(cid, "o=gauss,
                                               c=de, ou=groups", false);
Iterator grpIter = unassignedGroups.iterator();
while ( grpIter.hasNext() )
{
  StringValue grpDN = (StringValue) grpIter.next();
  ah.importLDAPPrincipalAsGroup(cid, grpDN);
}

...
```

4.3 Rechte von Principals bearbeiten

In VIP ContentManager wird zwischen zwei verschiedenen Berechtigungsarten unterschieden – den Administrations- und den Objektrechten. Die Administrationsrechte bestimmen, welche Benutzer Zugriff auf die Administrationsfunktionen von VIP ContentManager erhalten. Die Objektrechte bestimmen, welche Funktionen dem Benutzer beim Bearbeiten von VIP-Objekten zur Verfügung stehen. Allgemeine Informationen zu den Rechten erhalten Sie im Abschnitt 2.2 “Zugriffssteuerung” auf Seite 25.

Mithilfe der Funktionen des `AdminHandler`-Interfaces können Sie den Principals Administrationsrechte und Standard-Objektrechte zuweisen. Die Zugriffsrechte werden im VIP Java API durch folgende Interfaces repräsentiert:

- `VipAdminPermission` (siehe folgenden Abschnitt)
- `VipObjectPermission` (siehe “Standard-Objektrechte von Principals” ab Seite 86)

Administrationsrechte von Principals

Die folgende Tabelle zeigt die Konstanten, die im Interface `VipAdminPermission` für die einzelnen Administrationsrechte definiert sind. Die Rechte sind so aufgebaut, dass die umfassenderen Rechte andere Rechte einschließen. So beinhaltet z. B. das Recht “Principal ändern” das Recht “Zugriff auf Benutzerverwaltung”. Ausführliche Informationen zu den Funktionen, die den Administrationsrechten zugeordnet sind, bietet Tabelle 2 “Die einzelnen Administrationsrechte” auf Seite 30.

Tabelle 3 – Konstanten der Administrationsrechte

Administrationsrecht einschl. beinhalteter Rechte	Konstante
Lesender Zugriff auf Benutzerverwaltung (API) Beinhaltet “Zugriff auf Benutzerverwaltung”	`USER_MANAGEMENT_READ_ACCESS`
Zugriff auf Benutzerverwaltung	`USER_MANAGEMENT_ACCESS`
Principal ändern Beinhaltet “Zugriff auf Benutzerverwaltung”	`MODIFY_PRINCIPAL`
Zuordnung zu Website/Funktionsbereich ändern Beinhaltet “Zugriff auf Benutzerverwaltung”	`MODIFY_PRINCIPAL_ASSIGNMENT`
Principal anlegen, ändern, löschen Beinhaltet “Zugriff auf Benutzerverwaltung”, “Principal ändern”, “Zuordnung zu Website/Funktionsbereich ändern”	`CREATE_MODIFY_REMOVE_PRINCIPAL`
Administrationsrechte ändern Beinhaltet “Zugriff auf Benutzerverwaltung”	`SET_ADMIN_ACL`
Zugriff auf Konfiguration	`CONFIG_MANAGEMENT_ACCESS`
Konfigurationseintrag ändern Beinhaltet “Zugriff auf Konfiguration”	`MODIFY_CONFIG_ENTRY`

Administrationsrecht einschl. beinhalteter Rechte	Konstante
Konfigurationseintrag anlegen, ändern, löschen Beinhaltet “Zugriff auf Konfiguration”, “Konfigurationseintrag ändern”	CREATE_MODIFY_REMOVE_CONFIG_ENTRY
Zugriff auf Systemverwaltung	SYSTEM_MANAGEMENT_ACCESS
Systemstatus ändern Beinhaltet “Zugriff auf Systemverwaltung”	CHANGE_SYSTEM_STATE

Das Interface `VipAdminPermission` ist folgendermaßen aufgebaut:

```
public interface VipAdminPermission extends PermissionBase
{
 public final static VipAdminPermission USER_MANAGEMENT_READ_ACCESS;
 public final static VipAdminPermission USER_MANAGEMENT_ACCESS;
 public final static VipAdminPermission MODIFY_PRINCIPAL;
 public final static VipAdminPermission MODIFY_PRINCIPAL_ASSIGNMENT;
 public final static VipAdminPermission CREATE_MODIFY_REMOVE_PRINCIPAL;
 public final static VipAdminPermission SET_ADMIN_ACL;
 public final static VipAdminPermission CONFIG_MANAGEMENT_ACCESS;
 public final static VipAdminPermission MODIFY_CONFIG_ENTRY;
 public final static VipAdminPermission CREATE_MODIFY_REMOVE_
                                         CONFIG_ENTRY;
 public final static VipAdminPermission SYSTEM_MANAGEMENT_ACCESS;
 public final static VipAdminPermission CHANGE_SYSTEM_STATE;

 public String getName();
 public int getId();
}
```

Mithilfe der Methode `getAdminPermissions` des Interfaces `User`, `Group` bzw. `Role` können die Administrationsrechte eines Principals gelesen werden. Es wird eine Liste mit den im Interface `VipAdminPermission` definierten Konstanten zurückgeliefert, die als Administrationsrechte für den jeweiligen Principal gesetzt sind.

Um die Administrationsrechte eines Principals zu bearbeiten, wird die Methode `setAdminRights` des `AdminHandler`-Interfaces verwendet. In einer Liste werden die Konstanten für die Administrationsrechte übergeben.

Ein neu angelegter Benutzer verfügt zunächst nicht über Administrationsrechte. Das folgende Beispiel zeigt, wie einem Benutzer Administrationsrechte zugewiesen werden.

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//load profile of user ’Joshua Stein’
User usr = ah.getUserProfile(cid, "jstein",
                             AdminHandler.INDIRECT_ASSIGNMENTS);

//grant the administration rights MODIFY_PRINCIPAL and SET_ADMIN_ACL
//to the user
List admGranted = new LinkedList();
admGranted.add(VipAdminPermission.MODIFY_PRINCIPAL);
admGranted.add(VipAdminPermission.SET_ADMIN_ACL);
ah.setAdminPermissions(cid, usr, admGranted);
```

Standard-Objektrechte von Principals

Die Standard-Objektrechte werden beim Hinzufügen eines Principals zur Zugriffssteuerungsliste eines VIP-Objekts als Vorgaben für die Objektrechte übernommen. Die genauen Zugriffseinstellungen für ein VIP-Objekt gehören zu den Metadaten der Objekte und können über das Interface `ObjectHandler` geändert werden, siehe “Zugriffssteuerungslisten – das Interface ACL” ab Seite 206).

Das Interface `VipObjectPermission` stellt folgende Konstanten für die Objektrechte zur Verfügung:

Tabelle 4 – Konstanten der Zugriffsrechte für VIP-Objekte

Zugriffsrecht	Konstante
Lesen	`VipObjectPermission.READ`
Objekt ändern	`VipObjectPermission.WRITE`
Löschen	`VipObjectPermission.DELETE`
Anlegen	`VipObjectPermission.CREATE`
Freigeben	`VipObjectPermission.RELEASE`
Metadaten ändern	`VipObjectPermission.WRITE_META`
Rechte ändern	`VipObjectPermission.CHANGE_RIGHTS`
Verschieben und Kopieren	`VipObjectPermission.TREE_OPERATIONS`
Lesen (Produktion)	`VipObjectPermission.READ_PRODUCTION`

Das Interface `VipObjectPermission` ist folgendermaßen definiert:

```
public interface VipObjectPermission extends PermissionBase
{
    public final static VipObjectPermission READ;
    public final static VipObjectPermission WRITE;
    public final static VipObjectPermission DELETE;
    public final static VipObjectPermission CREATE;
    public final static VipObjectPermission RELEASE;
    public final static VipObjectPermission WRITE_META;
    public final static VipObjectPermission CHANGE_RIGHTS;
    public final static VipObjectPermission TREE_OPERATIONS;
    public final static VipObjectPermission READ_PRODUCTION;

    public String getName();
    public int getId();
}
```

Mithilfe der Methode `getInitialPrincipalRights` des Interfaces `User`, `Group` bzw. `Role` können die Standard-Objektrechte eines Principals gelesen werden. Es wird ein Objekt der Klasse `InitialPrincipalRights` zurückgeliefert. Diese Klasse stellt Methoden zum Abfragen und Setzen von erlaubten (granted) und verbotenen (denied) Objektrechten zur Verfügung:

```
public class InitialPrincipalRights
{
  public InitialPrincipalRights(Principal p, List grantedPermissions,
                                List deniedPermissions)
  public InitialPrincipalRights(List grantedPermissions,
                                List deniedPermissions)
  public Principal getPrincipal()
  public List getGrantedPermissions()
  public void setGrantedPermissions(List perm)
  public void setDeniedPermissions(List perm)
  public boolean hasGrantedPermission(VipObjectPermission p)
  public boolean hasDeniedPermission(VipObjectPermission p)
  public boolean hasUnspecifedPermission(VipObjectPermission p)
  public String toString()
}
```

Die Objektrechte, die mithilfe der Methoden von `InitialPrincipalRights` eingestellt werden, müssen anschließend als Standard-Objektrechte des Principals gespeichert werden. Dies erfolgt über die Methode `setInitialRights` des Interfaces `AdminHandler`.

Ein neu angelegter Benutzer verfügt zunächst nicht über Standard-Objektrechte. Das folgende Beispiel veranschaulicht das Setzen von Standard-Objektrechten für einen Benutzer:

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//load profile of user 'Joshua Stein'
User usr = ah.getUserProfile(cid, "jstein",
                             AdminHandler.INDIRECT_ASSIGNMENTS);

//read the default object rights of the user
InitialPrincipalRights initialRights = usr.getInitialRights();

//The rights CREATE, DELETE and READ is to be granted to the user
//These rights have to be created as list
List granted = new LinkedList();
granted.add(VipObjectPermission.CREATE);
granted.add(VipObjectPermission.DELETE);
granted.add(VipObjectPermission.READ);

//The rights WRITE and TREE_OPERATIONS are to be denied to the user
List denied = new LinkedList();
denied.add(VipObjectPermission.WRITE);
denied.add(VipObjectPermission.TREE_OPERATIONS);

//Set objects right lists at the InitialRights object
initialRights.setDeniedPermissions(denied);
initialRights.setGrantedPermissions(granted);

//Save the changed object rights with the user by calling
//setInitialRights at the AdminHandler
ah.setInitialRights(cid, usr, initialRights);
```

4.4 Funktionsbereiche bearbeiten

In VIP ContentManager müssen Benutzer direkt oder indirekt Funktionsbereichen zugeordnet sein, um bestimmte Aktionen z.B. im Rahmen der Website-Verwaltung durchführen zu können (siehe “Funktionsbereiche” auf Seite 27).

Im VIP Java API werden Funktionsbereiche durch das Interface `FunctionalArea` repräsentiert. Das `AdminHandler`-Interface erlaubt das Anlegen, Auslesen und Löschen von Funktionsbereichen. Lesen Sie dazu die folgenden Abschnitte:

- Konstanten für Funktionsbereiche (siehe folgenden Abschnitt)
- Funktionsbereiche bearbeiten (siehe “Funktionsbereiche anlegen und löschen” auf Seite 92)
- Funktionsbereiche auslesen (siehe “Funktionsbereiche des VIP-CM-Systems bestimmen” auf Seite 93)

Konstanten für Funktionsbereiche

Das Interface `FunctionalArea` enthält alle in VIP ContentManager vordefinierten Funktionsbereiche als Konstanten. Werden über das VIP-Administrationsprogramm oder das VIP Java API neue Funktionsbereiche angelegt, werden diese durch ihren Namen repräsentiert.

Einige Standard-Funktionsbereiche sind standardmäßig mit bestimmten Objekttypen verknüpft. Auf diese Weise können nur Benutzer mit dem entsprechenden Funktionsbereich Objekte dieses Typs anlegen. Die Verknüpfung eines Objekttyps mit einem Funktionsbereich kann mithilfe der Methode `getFunctionalArea` des `ObjectHandler`-Interfaces abgefragt werden. Die Verknüpfungen zwischen Objekttyp und Funktionsbereich können Sie nur über das VIP-Administrationsprogramm bearbeiten.

Funktionsbereiche können über das VIP-Administrationsprogramm oder das VIP Java API Principals zugeordnet werden. Auf diese Weise erhalten die Principals Zugriff auf die entsprechenden Funktionen zum Anlegen bestimmter Objekttypen bzw. Zugang zu den entsprechenden Funktionen des CMS-Clients. Informationen zum Zuordnen von Funktionsbereichen zu Principals enthält der Abschnitt 4.5 “Zuordnungen von Principals bearbeiten” ab Seite 94.

Die folgende Tabelle zeigt die Konstanten für die vordefinierten Funktionsbereiche einschließlich der abhängigen Funktionen oder Ansichten.

Tabelle 5 – Konstanten der Funktionsbereiche

Funktionsbereich	Konstante	Abhängige Ansicht oder Funktion im CMS-Client
Anlegen Basis	`VIP`	Anlegen von Objekten, die auf den zugeordneten Objekttypen basieren
Anlegen Fortgeschritten	`ADVANCED`	
Anlegen Dynamisch	`DYNAMIC`	
Anlegen Formular	`FORM`	
Anlegen Personalisierung	`PERSONALIZATION`	
Intelligente Vorlagen	`ITF`	Wird nicht standardmäßig verwendet, dient der Abwärtskompatibilität zu VIP 5e
Direkte Freigabe	`DIRECT_RELEASE`	Bearbeiten der Option *Direkte Freigabe* (Metadaten-Dialog)
Dialog Referenzen	`REFERENCES`	Ansicht des Referenzen-Dialogs

Funktionsbereich	Konstante	Abhängige Ansicht oder Funktion im CMS-Client
Dialog Zugriffsrechte	ACCESS_RIGHTS	Ansicht des Zugriffsrechte-Dialogs
Dialog Protokoll	LOG	Ansicht des Protokoll-Dialogs
Filter Standard	FILTER_STANDARD	Verwenden der Standardfilter
Filter Bearbeiten	FILTER_EDIT	Bearbeiten von Filtern im Filtereditor
Ansicht Untergeordnete Objekte	OBJECTLIST	Ansicht “Untergeordnete Objekte”
Ansicht Objektliste	LISTVIEW	Ansicht “Objektliste”
Ansicht Meine Objekte	FILTER_TODO	Ansicht “Meine Objekte”
Ansicht Vorlagenstruktur	TEMPLATE_STRUCTURE	Ansicht “Vorlagenstruktur”
Importieren	IMPORT	Verwenden der Importfunktionen
Volltextsuche	COMI_SEARCH	Verwenden der Suchfunktionen

Das Interface `FunctionalArea` ist folgendermaßen definiert:

```
public interface FunctionalArea extends Value
{
    public static final String VIP ;
    public static final String ADVANCED ;
    public static final String DIRECT_RELEASE ;
    public static final String DYNAMIC ;
    public static final String FORM ;
    public static final String ITF ;
    public static final String PERSONALIZATION ;
    public static final String REFERENCES ;
```

```
    public static final String ACCESS_RIGHTS ;
    public static final String LOG ;
    public static final String OBJECTLIST ;
    public static final String FILTER_STANDARD ;
    public static final String FILTER_TODO ;
    public static final String FILTER_EDIT ;
    public static final String LISTVIEW ;
    public static final String TEMPLATE_STRUCTURE ;
    public static final String IMPORT ;
    public static final String COMI_SEARCH ;

    public boolean hasAssignment ();
    public String getName ();
    public List getUsers ();
    public List getGroups ();
    public List getRoles ();
}
```

Funktionsbereiche anlegen und löschen

Das Anlegen von Funktionsbereichen erfolgt über die FunctionalAreaFactory. Die Factory dient lediglich zum Erstellen der Funktionsbereich-Objekte. Um die neu erstellten Funktionsbereiche dauerhaft dem VIP-CM-System hinzuzufügen, muss anschließend die add-Methode des AdminHandler-Interfaces aufgerufen werden.

Das Interface FunctionalAreaFactory verfügt nur über eine Methode:

```
public interface FunctionalAreaFactory
{
 public FunctionalArea createFunctionalArea(String name);
}
```

Eine Instanz der FunctionalAreaFactory kann über die Methode getFunctionalAreaFactory des AdminHandler-Interfaces angefordert werden.

Beim Anlegen eines Funktionsbereichs über die Methode `createFunctionalArea` wird der Name des neuen Funktionsbereichs als String übergeben.

Das Löschen von Funktionsbereichen erfolgt mithilfe der Methode `remove` des `AdminHandler`-Interfaces.

Funktionsbereiche des VIP-CM-Systems bestimmen

Um die innerhalb eines VIP-CM-Systems verwendeten Funktionsbereiche sowie deren spezifische Eigenschaften zu bestimmen, können die Methoden `getFunctionalArea` und `getFunctionalAreas` des `AdminHandler`-Interfaces verwendet werden.

Die Methode `getFunctionalAreas` liefert eine Liste von `FunctionalArea`-Objekten zurück, allerdings ohne deren Eigenschaften, d. h., die Benutzer, Gruppen und Rollen sind nicht definiert. In diesem Fall liefert die Methode `hasAssignment` den Wert `false` zurück. Über die Methode `getFunctionalArea` kann ein vollständig geladenes Objekt vom Administrationsserver erfragt werden (`hasAssignment` liefert dann den Wert `true` zurück).

4.5 Zuordnungen von Principals bearbeiten

Für die Principals des VIP-CM-Systems können verschiedene Zuordnungen vorgenommen werden:

- Zuordnen von Benutzern zu Gruppen bzw. Rollen

  Die Benutzer erhalten damit die Rechte der entsprechenden Gruppen bzw. Rollen.

- Zuordnen von Benutzern als Stellvertreter anderer Benutzer

  Ein Stellvertreter erhält nach entsprechender Anmeldung die Objektrechte des anderen Benutzers.

- Zuordnen von Principals zu Websites

  Wenn Sie einem Principal eine Website zuordnen, können die entsprechenden Benutzer auf die Objekte der Website zugreifen.

- Zuordnen von Principals zu Funktionsbereichen

  Über die Funktionsbereiche steuern Sie, welche Typen von Objekten die Benutzer anlegen dürfen und welche Dialoge und Funktionen im CMS-Client zur Verfügung stehen.

Zur Bearbeitung von Zuordnungen stellt das `AdminHandler`-Interface die Methoden `defineAssignment` und `removeAssignment` zur Verfügung.

Zuordnen von Principals zu Gruppen/Rollen, Websites und Funktionsbereichen

Das folgende Beispiel zeigt, wie ein Benutzer einer Gruppe zugeordnet wird. Anschließend erfolgt die Zuordnung des Benutzers zu einer Website und zu allen vorhandenen Funktionsbereichen.

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler aH = VipRuntime.getAdminHandler();

//load user 'jstein' and group 'Administration'
//assign user 'jstein' to this group
User usr = aH.getUserProfile(cid, "jstein",
                             AdminHandler.INDIRECT_ASSIGNMENTS);
Group grpAdmin = aH.getGroupProfile(cid, "Administration");
aH.defineAssignment(cid, usr, grpAdmin);

//assign the user to website "SDF"
Website sdf = aH.getWebsite(cid, "SDF", AdminHandler.NO_ASSIGNMENTS);
aH.defineAssignment(cid, sdf, usr);

//assign all available functional areas to the user
List faList = AdminHandler.getFunctionalAreas(cid);
Iterator faIter = faList.iterator();
while ( faIter.hasNext() )
{
FunctionalArea fa = (FunctionalArea)faIter.next();
ah.defineAssignment(cid, fa, usr);
}
```

Hinweis: Um festzustellen, welche Principals einer Website, einem Funktionsbereich oder einer Gruppe/Rolle zugeordnet sind, nutzen Sie die entsprechenden `get`-Methoden der Interfaces `Website` (z.B. `getUsers()`), `FunctionalArea` (z.B. `getGroups()`), `User` (z.B. `getRoles()`) oder `Group` bzw. `Role` (`getMembers()`).

Stellvertreter-Zuordnungen

Wenn Sie die Zuordnungen zwischen Principals bearbeiten und in der Methode `defineAssignment` bzw. `removeAssignment` die Kennungen von zwei Benutzern als Argumente übergeben, wird der erste Benutzer als Stellvertreter des zweiten Benutzers eingerichtet.

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler aH = VipRuntime.getAdminHandler();

//The user "mmoreno" is to be assigned as substitute for "jstein"
User usrJst = aH.getUserProfile(cid, "jstein",
              AdminHandler.INDIRECT_ASSIGNMENTS);
User substitute = aH.getUserProfile(cid, "mmoreno",
                  AdminHandler.INDIRECT_ASSIGNMENTS);

//The first user specified in the parameters of the method
//defineAssignment becomes the substitute of the second user
aH.defineAssignment(cid, substitute, usrJst );
```

4.6 Informationen zu VIP-CM-Servern

Ein VIP-CM-System besteht immer aus einer Menge von VIP-CM-Servern, die miteinander kommunizieren. Die Serverkategorien und -typen sind im Abschnitt “Workflow und Datenhaltungssichten” auf Seite 21 erläutert.

Jeder VIP-CM-Server wird eindeutig durch einen logischen Namen identifiziert und hat einen Typ, der die Rolle des Servers definiert (Edit, QS, Produktion etc.). Das Interface `Server` definiert alle verwendeten Servertypen. Der Name und der Typ des Servers können ausgelesen werden.

Weiterhin kann mit `isMaster` überprüft werden, ob es sich um einen Master-Server handelt (Master-Edit- oder Master-Administrationsserver). Nur ein Master-Server hat schreibenden Zugriff auf die Daten des VIP-CM-Systems. Liefert die Methode `false` zurück, handelt es sich um einen Proxy-Server. Ein Proxy-Server kann nur lesend auf die VIP-Objekte einer Website zugreifen und delegiert alle schreibenden Aktionen an einen Master-Server.

Über das `AdminHandler`-Interface können detaillierte Serverinformationen ermittelt werden. Dazu wird die Methode `getServer` verwendet, die ein `Server`-Objekt zurückliefert. Folgende Servereigenschaften sind dort enthalten:

- der für die Benutzerverwaltung und Systemkonfiguration verantwortliche Administrationsserver
- Art der Speicherung der Benutzerdaten (RDBMS oder LDAP)
- der Host-Name des Servers sowie die für die Kommunikation verwendeten Port-Adressen
- der Host-Name des Mailservers, der für die Versendung von automatischen Benachrichtigungen verwendet wird, sowie die E-Mail-Adresse des dafür verwendeten Absenders
- die vom Server angebotenen Sprachen für die Lokalisierung von Texten
- die Liste aller Websites, die von diesem Server verwaltet werden bzw. auf die dieser Server lesend zugreifen kann (im Falle von Proxy-Servern)
- die Liste der Deploymentsysteme, die auf diesem Server installiert sind
- das Installationsverzeichnis sowie das temporäre Verzeichnis des Servers. Unterhalb des temporären Verzeichnisses des Servers gibt es für jede Website ein temporäres Verzeichnis, das im Rahmen der Objektverwaltung benötigt wird (z.B. für das Ausleihen von VIP-Objekten).

```
public interface Server extends Value
{
   public static final String TYPE_EDIT;
   public static final String TYPE_QA;
   public static final String TYPE_PROD;
   public static final String TYPE_PORTALMANAGER;
   public static final String TYPE_ADMIN;

   public String getName();
   public String getInstallDirectory();
   public String getTempDirectory();
   public String getType();
   public boolean isMaster();

   public String getHostname();
   public int getSocketPort();
   public int getHTTPPort();
   public String getMailHost();
   public List getLanguages();

   public Server getAdminServer();

   public List getWebsite(String websiteName);
   public Website getWebsiteNames();

   public List getDeploymentSystems();
   public DeploymentSystem getDeploymentSystem(String dplName);

   public boolean getSecure();

}
```

4.7 Informationen zu Websites

Eine Website wird innerhalb eines VIP-CM-Systems eindeutig durch einen logischen Namen identifiziert. Das `AdminHandler`-Interface bietet Methoden, um die einer Website zugeordneten Informationen auszulesen. Eine Website wird durch ein Objekt der Klasse `Website` modelliert. Die Methode `getWebsite` des `AdminHandler`-Interfaces liefert analog zur Benutzerverwaltung die wesentlichen Eigenschaften der Website. Die Liste aller Websites innerhalb des VIP-CM-Systems wird mit `getWebsites` ermittelt. Dies ist eine Liste von `Website`-Objekten.

Zu einem `Website`-Objekt können folgende Eigenschaften ermittelt werden:

- das Wurzelobjekt der Website, d.h. das VIP-Objekt, das die Wurzel des Navigationsbaums repräsentiert
- die Menge aller Benutzer, Rollen und Gruppen, die dieser Website zugeordnet sind. Nur Benutzer, die über eine entsprechende Zuordnung verfügen (eine direkte Zuordnung als Benutzer oder eine indirekte aufgrund der Gruppen- bzw. Rollenzugehörigkeit), haben Zugriff auf diese Website. Die Mengen enthalten entsprechende Objekte der Klassen `User`, `Role` und `Group`.
- der Name des Master-Edit-Servers der Website sowie der Name des JDBC-Pools, der vom Master-Edit-Server zu Speicherung der Website-Daten verwendet wird
- eine Liste der Proxy-Server (`String`-Objekte), die lesend auf die Website zugreifen können (und schreibende Zugriffe delegieren)
- eine Liste aller Objekttypen, die für diese Website definiert sind. Die Liste enthält Objekte des Interfaces `ObjectType`.
- eine Liste aller Objektstatus mit Objekten des Interfaces `ObjectState`

- das Encoding, das für die Generierung der Web-Objekte verwendet wird (UTF-8 oder ISO-8859-1)
- eine Liste der Deploymentsysteme (`String`-Objekte) dieser Website
- die Routinginformation sowie die verwendete Datenhaltung des Master-Edit-Servers
- das temporäre Verzeichnis der Website. Dieses Verzeichnis wird für Dateien verwendet, die vom Server zu schreiben sind, z.B. um den Inhalt eines ausgeliehenen VIP-Objekts zu speichern.

```
package de.gauss.vip.api.admin;

public interface Website
{
   public File getTemporaryDirectory();
   public ObjectId getNavigationRoot();
   public String getName();
   public boolean hasAssignment();
   public List getRoles();
   public List getUsers();
   public List getGroups();
   public List getObjectTypes() throws VipApiException;
   public ObjectType getObjectType(String name) throws VipApiException;
   public List getObjectStates() throws VipApiException;
   public ObjectState getObjectState(String name) throws
      VipApiException;
   public List getDeploymentSystems(String serverName)
      throws VipApiException;
   public DeploymentSystem getDeploymentSystem(String dplName)
      throws VipApiException;
   public String getServerName();
   public String getWebObjectCharacterEncoding();
   public String getPoolName();
   public Map getProxyServers();
   public Map getRoutingTable();
}
```

4.8 Informationen zu Deploymentsystemen

Es gibt drei verschiedene Typen von Deploymentsystemen, entsprechend den drei Workflow-Schritten Edit, QS und Produktion. Abhängig vom Typ erzeugt ein Deploymentsystem eine bestimmte Sicht auf die VIP-Objekte der Website. Ein Edit-Deploymentsystem verfügt über alle VIP-Objekte und verwendet diese für die Seitengenerierung. Im QS- bzw. Produktionsdeploymentsystem sind nur die entsprechenden Daten der jeweiligen Sicht vorhanden.

Außerdem unterscheiden sich Deploymentsysteme durch ihre Kategorie: *Standard*, *dynamisch*, *WebDAV* oder *Suchmaschinen*. Die Kategorie bestimmt die Art der Verarbeitung von Deploymentaufträgen. Weitere Hinweise zu den Typen und Kategorien von Deploymentsystemen erhalten Sie im VIP ContentManager-Administratorhandbuch.

Die wesentlichen Eigenschaften eines Deploymentsystems sind:

- Name
- Typ (`EDIT`, `QA`, `PROD`)
- Kategorie (`STANDARD`, `DYNAMIC`, `WEBDAV` und `CONTENT` für Suchmaschinen-Deploymentsysteme)
- Website, auf der das Deploymentsystem basiert
- Server, auf dem das Deploymentsystem ausgeführt wird
- Verzeichnis auf dem Server, in dem die generierten Seiten gespeichert werden sollen
- Basis-URL, die für die Generierung von Verweisen verwendet werden soll

Über die Methode `getDeploymentSystems` des `AdminHandler`-Interfaces kann die Liste aller Deploymentsysteme ermittelt werden.

```
package de.gauss.vip.api.admin
public interface DeploymentSystem
{
    public static String TYPE_EDIT;
    public static String TYPE_QA;
    public static String TYPE_PROD;

    public static String CAT_STANDARD:
    public static String CAT_DYNAMIC;
    public static String CAT_WEBDAV;
    public static String CAT_CONTENT;

    public int getMaxThreads();
    public int getMinThreads();
    public String getName();
    public String getType();
    public String getCategory();
    public String getWebsiteName();
    public String getServerName();
    public String getBaseURL();
    public String getBaseDirectory();
    public boolean getRootTopicNeedsDirectory();
}
```

4.9 Runlevels

Der Runlevel bezeichnet den aktuellen Zustand eines VIP-CM-Servers oder einer Website. Er definiert, welche funktionalen Komponenten des Servers oder der Website gerade aktiviert bzw. deaktiviert sind. Die Runlevels sind unterteilt in Server-Runlevels und Website-Runlevels, wobei die Website-Runlevels auf den Server-Runlevels aufbauen. Weitere Angaben zu den Runlevels finden Sie im VIP ContentManager-Administratorhandbuch.

Die möglichen Zustände eines VIP-CM-Servers bzw. einer Website sind im Interface `Runlevel` definiert. Beim Start eines Servers werden diese Zustände von Runlevel `SERVER_DOWN` bis Runlevel `WEBSITE_UP` durchlaufen, beim Herunterfahren des Servers entsprechend von `SERVER_UP` bis `SERVER_DOWN`. Darüber hinaus können die Runlevels von Servern und Websites separat gesteuert werden.

Folgende Runlevels von VIP-CM-Servern gibt es:

- **`SERVER_DOWN`**: Der Server ist heruntergefahren.
- **`CONNECTIONS_CLOSED`**: Alle Kommunikationsverbindungen wurden geschlossen. Datenbank- und LDAP-Verbindungen sowie Dienste sind nicht mehr verfügbar. Beim Übergang zum nächsthöheren Runlevel wird die Verbindungsverwaltung aufgebaut.
- **`USERS_LOGGED_OUT`**: Es sind keine Benutzer mehr an diesem VIP-CM-Server angemeldet. Beim Übergang zum nächsthöheren Runlevel wird die Benutzerverwaltung aufgebaut.
- **`AGENTS_STOPPED`**: Alle Server-Agenten wurden beendet. Das VIP Java API kann daher erst ab dem nächsthöheren Runlevel verwendet werden. Beim Übergang zum nächsthöheren Runlevel werden alle Server-Agenten gestartet.
- **`SINGLE_USER`**: Einzelnutzerbetrieb. Alle Benutzer bis auf den Administrator, der die Runlevel-Änderung vornimmt, werden abgemeldet und können sich nicht mehr am VIP-CM-System anmelden. Dieser Runlevel ist besonders für Wartungsarbeiten am VIP-CM-System vorgesehen. Beim Übergang zum Runlevel `AGENTS_STOPPED` werden die Websites vollständig heruntergefahren. Beim Übergang zum nächsthöheren Runlevel wird das System für alle Benutzer freigegeben.
- **`SERVER_UP`**: Der VIP-CM-Server ist vollständig hochgefahren.

Folgende Runlevels von Websites gibt es. Für die Website-Runlevels wird zumindestens der Server-Runlevel `SINGLE_USER` vorausgesetzt.

- **WEBSITE_INACCESSIBLE**: Eine bzw. alle Websites sind nicht mehr verfügbar (auch nicht für lesende Zugriffe). Damit sind auch die Deploymentsysteme nicht mehr aktiv. Die entsprechenden Ereignisse des `Event`-Interfaces (`RUNLEVEL_INCREASES` bzw. `RUNLEVEL_DECREASES`) enthalten als Argument die Website bzw. `null` für alle Websites.
- **WEBSITE_CONFIGURATION**: Die Konfigurationsdaten der Website werden geschrieben. Beim Übergang zum nächsthöheren Runlevel werden die Websites für den lesenden Zugriff initialisiert.
- **WEBSITE_READONLY**: Auf eine bzw. auf alle Websites kann lesend zugegriffen werden. Das VIP-Ereignis hat die gleichen Argumente wie im Runlevel `WEBSITE_INACCESSIBLE`. Beim Übergang zum nächsthöheren Runlevel werden die Deploymentsysteme initialisiert.
- **DEPLOYMENT_COMPLETED**: Die Initialisierung der Deploymentsysteme ist abgeschlossen. Die Deploymentsysteme können nun ausstehende Aufträge ausführen. Neue Aufträge werden entgegengenommen und die entsprechenden Seiten bearbeitet. Beim Übergang zum nächsthöheren Runlevel werden die Websites für den schreibenden Zugriff freigegeben. Die Verteilung von VIP-Objekten an Proxy-Server findet erst im nächsten Runlevel statt.
- **WEBSITE_UP**: Eine bzw. alle Websites stehen für den schreibenden Zugriff zur Verfügung. Auch hier enthalten die entsprechenden Ereignisse des `EventDispatcher`-Interfaces als Argument die Website bzw. `null` für alle Websites. Wenn also das Event `RUNLEVEL_IS` mit dem Runlevel `WEBSITE_UP` gefeuert wird und als Argument für den Website-Namen `null` geliefert wird, heißt das, alle Websites sind vollständig gestartet.

```
package de.gauss.vip.api.admin
public interface Runlevel
{
        public static final int SERVER_DOWN ;
        public static final int CONNECTIONS_CLOSED ;
        public static final int USERS_LOGGED_OUT ;
        public static final int AGENTS_STOPPED ;
        public static final int SINGLE_USER ;
        public static final int SERVER_UP ;

        public static final int WEBSITE_INACCESSIBLE ;
        public static final int WEBSITE_CONFIGURATION ;
        public static final int WEBSITE_READONLY ;
        public static final int DEPLOYMENT_COMPLETED ;
        public static final int WEBSITE_UP ;
}
```

Das Administrationsinterface `AdminHandler` bietet Methoden an, um den Runlevel des aktuellen Servers bzw. der Website zu ändern, z.B. um das System soweit herunterzufahren, bis ein Backup einer Website möglich ist. Mit `getRunlevel` wird der Runlevel des Servers bzw. der Website ermittelt. Mit `setRunlevel` wird dieser entsprechend geändert. Dies ist i.d.R. ein längerer Prozess (ähnlich dem Starten bzw. Herunterfahren eines Betriebssystems).

> **Hinweis:** Informationen zu den Ereignissen, die bei Runlevel-Änderungen gefeuert werden, erhalten Sie im Kapitel 5 “Ereignisverarbeitung”.

KAPITEL 5

Ereignisverarbeitung

Bei statusverändernden Aktionen in einem VIP-CM-System werden Ereignisse gefeuert. Dazu gehören sowohl *Ereignisse*, die nach der Aktion gefeuert werden (wenn die Statusveränderung erfolgt ist), als auch *Vorbereitungsereignisse*, die vor der eigentlichen Aktion gefeuert werden.

Um über das Eintreten von bestimmten Ereignissen eine Benachrichtigung zu erhalten, können Server-Agenten einen *Ereignisbeobachter* (`EventListener` bzw. `DeploymentEventListener`) am *Ereignisverteiler* (`EventDispatcher` bzw. `DeploymentEventDispatcher`) registrieren. Beim Auftreten eines Ereignisses verteilt der zentrale Ereignisverteiler das Ereignis an alle Ereignisbeobachter, die sich für dieses Ereignis registriert haben.

Grundsätzlich hat jedes Ereignis einen eindeutigen Typ, der die Art der Statusänderung (bzw. der bevorstehenden Änderung) repräsentiert. Die möglichen Ereignistypen sind als Konstanten in den Interfaces `Event`, `DeploymentEvent` und `PrepareEvent` definiert.

> **Hinweis:** Ereignisse werden unabhängig vom Typ des Servers auf dem erzeugenden VIP-CM-Server geworfen. Website- und System-bezogene Events werden auf dem Master-Edit-Server ausgelöst und sollten auch dort verarbeitet werden. Vorbereitungsereignisse sind grundsätzlich nur auf dem Master-Edit-Server verfügbar.

Ein wichtiger Unterschied zwischen den Vorbereitungsereignissen und den “normalen” Ereignissen (Interfaces `Event` und `DeploymentEvent`) liegt darin, dass Vorbereitungsereignisse nur Zugriff auf die Benutzerkennung des jeweiligen Benutzers haben, der die entsprechende Aktion ausgeführt hat. Hingegen erhalten “normale” Ereignisse ein Authentifizierungsobjekt (`ContextId`), das es ihnen erlaubt, unter dem angemeldeten Benutzerkontext weitere Aktionen durchzuführen. Das bedeutet:

- “Normale” Ereignisse verfügen über alle Rechte des Benutzers, der die zugehörige Aktion ausgeführt hat.
- Vorbereitungsereignisse können keine Änderungen an der eigentlichen Aktion durchführen.

5.1 Ereignisse

In einem VIP-CM-System werden Ereignisse gefeuert, wenn sich der Status eines VIP-Objekts ändert, ein Web-Objekt verarbeitet wird oder sich der Zustand des gesamten Systems ändern.

Dementsprechend gehören Ereignisse zu verschiedenen Kategorien:

- Website-bezogene Ereignisse, die sich auf Statusveränderungen von VIP-Objekten einer Website beziehen
- System-bezogene Ereignisse, die sich auf eine Zustandsveränderung des VIP-CM-Systems beziehen
- Deployment-bezogene Ereignisse, die sich auf Statusveränderungen von Web-Objekten beziehen

Website- und System-bezogene Ereignisse werden durch Objekte des Interfaces `Event` repräsentiert, Deployment-bezogene Ereignisse durch das Interface `DeploymentEvent`.

Das Interface `Event`

Nach einem erfolgreichen Statuswechsel werden Ereignisse gefeuert, die durch Objekte des Interfaces `Event` repräsentiert werden.

Website-Ereignisse beziehen sich immer auf genau eine Website. Der Name dieser Website (als `String`-Objekt) kann mit der Methode `getConstraint` ermittelt werden. Für System-bezogene Ereignisse (z.B. ein Ereignis darüber, dass eine neue Website angelegt wurde) liefert diese Methode immer `null` zurück.

Jedes Ereignis hat Argumente, die das Ziel der Statusveränderung näher identifizieren. So haben z.B. alle Website-bezogenen Ereignisse eine Referenz auf das betreffende VIP-Objekt als Argument (OID). Grundsätzlich können die Werte der vom Ereignistyp abhängigen Argumente mit der Methode `getArgument` ermittelt werden. Alle Argumente können auch als Objekt der Behälter-Klasse `java.util.Map` verarbeitet werden.

```
public interface Event extends Serializable
{
   public int getId();
   public Object getConstraint();
   public ContextId getContextId();
   public Map getArguments();
   public Object getArgument(String argumentKey);
   public long getTID();
}
```

Website-bezogene Ereignisse

Die in diesem Abschnitt beschriebenen Ereignisse beziehen sich auf die VIP-Objekte einer Website. Für diese Kategorie von Ereignissen liefert die Methode `getConstraint` des `Event`-Interfaces den Namen der Website (als `String`-Objekt). Jedes Ereignis enthält als Argument die OID des betroffenen VIP-Objekts. Dafür wird die Konstante `Event.ARG_OID` verwendet. Der Zugriff erfordert dann einen entsprechenden Cast, z.B.

```
ObjectId oid = (ObjectId)event.getArgument(ARG_OID);
```

In der Beschreibung wird [in Klammern] die Methode des `ObjectHandler`-Interfaces angegeben, die das entsprechende Ereignis verursacht.

OBJECT_ACL_CHANGED

Dieses Ereignis wird gefeuert, wenn sich die Zugriffssteuerungsliste (ACL) eines VIP-Objekts geändert hat. Wenn sich nur die Zugriffssteuerungsliste geändert hat, wird das Ereignis `OBJECT_METADATA_CHANGED` nicht gefeuert! [`change`]

Argument

- `ARG_OID [ObjectId]`

OBJECT_CHECKED_IN

Dieses Ereignis wird gefeuert, wenn ein ausgeliehenes VIP-Objekt zurückgegeben wurde. [`checkIn`]

Argument

- `ARG_OID [ObjectId]`

OBJECT_CHECKED_OUT

Dieses Ereignis wird gefeuert, wenn ein VIP-Objekt ausgeliehen wurde. [`checkOut`]

Argument

- `ARG_OID [ObjectId]`

`OBJECT_CHECKOUT_UNDONE`

Dieses Ereignis wird gefeuert, wenn das Ausleihen eines Objekts rückgängig gemacht wurde. [`undoCheckOut`]

Argument

- `ARG_OID [ObjectId]`

`OBJECT_CREATED`

Dieses Ereignis wird gefeuert, wenn ein neues VIP-Objekt angelegt wurde. [`create`]

> **Hinweis:** Dieses Ereignis wird gefeuert, **bevor** das Deployment für das neue Objekt startet. Aus diesem Grunde sind vom Deployment generierte Metadaten wie z.B. `Pathname` oder `URL` nicht sofort verfügbar.

Argumente

- `ARG_OID [ObjectId]`
- `ARG_POID [ObjectId]`

  OID des Themas, in dem das VIP-Objekt angelegt wurde

`OBJECT_DELETED`

Dieses Ereignis wird gefeuert, wenn ein VIP-Objekt gelöscht wurde. [`delete`]

Argument

- `ARG_OID [ObjectId]`

OBJECT_DESTROYED

Dieses Ereignis wird gefeuert, wenn ein VIP-Objekt endgültig zerstört wurde. [destroy].

Argument

- ARG_OID [ObjectId]

OBJECT_EXPIRED

Dieses Ereignis wird gefeuert, wenn das Ablaufdatum eines VIP-Objekts erreicht ist und vom Agenten “Ablaufbenachrichtigung” eine Mail über das Ablaufen des Objekts verschickt wird. In der Konfiguration dieses Agenten kann festgelegt werden, wie oft der Agent überprüfen soll, welche Objekte abgelaufen sind. Davon hängt es ab, wann und wie oft dieses Ereignis gefeuert wird.

Argument

- ARG_OID [ObjectId]

OBJECT_METADATA_CHANGED

Dieses Ereignis wird gefeuert, wenn die Metadaten eines VIP-Objekts geändert wurden. [change]

Argumente

- ARG_OID [ObjectId]
- ARG_OLD [Version]

  Version des VIP-Objekts vor der Änderung

OBJECT_REJECTED_TO_EDIT

Dieses Ereignis wird gefeuert, wenn ein vorgelegtes VIP-Objekt abgelehnt wurde. [reject]

Argument

- ARG_OID [ObjectId]

OBJECT_SUBMIT_AT_TO_PRODUCTION

Dieses Ereignis wird gefeuert, wenn ein VIP-Objekt erst zu einem späteren Zeitpunkt freigegeben wird (verzögerte Freigabe). [release]

Argumente

- ARG_OID [ObjectId]
- ARG_DATE [Date]

  Zeitpunkt der Freigabe

OBJECT_SUBMITTED_TO_PRODUCTION

Dieses Ereignis wird gefeuert, wenn ein vorgelegtes VIP-Objekt für die Produktionssicht freigegeben wurde. [release]

Argument

- ARG_OID [ObjectId]

OBJECT_SUBMITTED_TO_QA

Dieses Ereignis wird gefeuert, wenn ein VIP-Objekt zur Qualitätssicherung vorgelegt wurde. [submit]

Argument

- ARG_OID [ObjectId]

System-bezogene Ereignisse

Die System-bezogenen Ereignisse beziehen sich nicht auf die einzelnen VIP-Objekte einer Website, sondern auf das VIP-CM-System als Ganzes. Bei diesen Ereignissen gibt es daher kein `Constraint`-Argument, das den Namen der Website enthält. Die Methode `getConstraint` des `Event`-Interfaces liefert immer `null` zurück.

RUNLEVEL_DECREASES

Dieses Ereignis wird gefeuert, wenn sich der Runlevel des aktuellen VIP-CM-Servers bzw. einer Website verringert hat.

Argumente

- `ARG_NEW [Integer]`

  Runlevel nach der Aktion, muss kleiner als der Runlevel davor sein

- `ARG_OLD [Integer]`

  Runlevel vor der Aktion

- `ARG_WEBSITE [String]`

  Website-Name oder `null`, falls sich die Runlevel-Änderung nicht auf eine Website bezogen hat

RUNLEVEL_INCREASES

Dieses Ereignis wird gefeuert, wenn sich der Runlevel des aktuellen VIP-CM-Servers bzw. einer Website erhöht hat.

Argumente

- `ARG_NEW [Integer]`

  Runlevel nach der Aktion, muss höher als der Runlevel davor sein

- `ARG_OLD [Integer]`

  Runlevel vor der Aktion

- `ARG_WEBSITE [String]`

  Website-Name oder `null`, falls sich die Runlevel-Änderung nicht auf eine Website bezogen hat

`RUNLEVEL_IS`

Dieses Ereignis wird gefeuert, wenn sich der Runlevel des aktuellen VIP-CM-Servers bzw. einer Website geändert hat.

Informationen zu den einzelnen Runlevels erhalten Sie im Abschnitt 4.9 “Runlevels” auf Seite 102.

Argumente

- `ARG_NEW [Integer]`

  Runlevel nach der Aktion

- `ARG_OLD [Integer]`

  Runlevel vor der Aktion

- `ARG_WEBSITE [String]`

  Website-Name oder `null`, falls sich die Runlevel-Änderung nicht auf eine Website bezogen hat

`USER_LOGIN`

Dieses Ereignis wird gefeuert, wenn sich ein Benutzer am aktuellen VIP-CM-Server anmeldet.

Argument

- `ARG_NAME [String]`

  Benutzerkennung (User-ID)

USER_LOGOUT

Dieses Ereignis wird gefeuert, wenn sich ein Benutzer am aktuellen VIP-CM-Server abmeldet.

Argument

- `ARG_NAME [String]`

  Benutzerkennung (User-ID)

WEBSITE_DELETED

Dieses Ereignis wird gefeuert, wenn eine Website gelöscht wurde. Nachdem eine Website gelöscht wurde, ist ein Zugriff auf die Website nicht mehr möglich. Um rechtzeitig auf das Löschen einer Website zu reagieren, muss man sich auf das Ereignis `RUNLEVEL_IS` registrieren `(Runlevel WEBSITE_INACCESSIBLE)`.

Argument

- `ARG_NAME [String]`

  Name der gelöschten Website

WEBSITE_NEW

Dieses Ereignis wird gefeuert, sobald eine neue Website angelegt wurde. Erst danach kann auf die VIP-Objekte der Website zugegriffen werden. Ob die VIP-Objekte gelesen oder geändert werden können, hängt von dem Runlevel der Website ab (siehe `RUNLEVEL_IS`).

Argument

- `ARG_NAME [String]`

  Name der neuen Website

Das Interface `DeploymentEvent`

Deployment-Ereignisse informieren über Veränderungen im Deployment. Es gibt zwei Arten von Ereignissen:

1. Ereignisse, die den Status des Deploymentsystems betreffen (Anlegen und Löschen)

2. Ereignisse, die den Status der Web-Objekte des Deploymentsystems betreffen (Anlegen, Ändern und Löschen).

   Diese Ereignisse beziehen sich immer auf persistent gespeicherte Web-Objekte (im Rahmen des dynamischen Deployments kann es auch temporär erzeugte Web-Objekte geben). Sie sind Folge einer über Ereignisse vermittelten Statusänderung des betreffenden VIP-Objekts oder eines `GENERATE_PAGE`-Events.

Alle Ereignisse werden gefeuert, nachdem die betreffende Änderung erfolgt ist. Ein wichtiger Unterschied zu Website- und System-bezogenen Ereignissen besteht darin, dass es beim Deployment keine Vorbereitungs-ereignisse gibt. Der Grund dafür ist, dass das Deployment nur die Repräsentationen der VIP-Objekte erzeugt, jedoch selbst keine Veränderungen an den VIP-Objekten vornimmt. Gegebenenfalls auftretende Konflikte werden vom Deployment selbst gelöst. Die Erzeugung der Web-Objekte wird optimiert, sodass bei Änderungen eines VIP-Objekts das betroffene Web-Objekt möglichst nur einmal generiert wird.

Deployment-Ereignisse werden durch Objekte des Interfaces `DeploymentEvent` repräsentiert und beziehen sich immer auf genau ein Deploymentsystem. Der Name dieses Deploymentsystems (als `String`-Objekt) kann mit der Methode `getConstraint` ermittelt werden.

Jedes Ereignis hat Argumente, die das Ziel der Statusveränderung näher identifizieren. So haben z.B. alle Deployment-bezogenen Ereignisse eine Referenz auf das betreffende VIP-Objekt als Argument (OID). Grundsätzlich können die Werte der vom Ereignistyp abhängigen Argumente mit der Methode `getArgument` ermittelt werden. Alle Argumente können auch als Objekt der Behälter-Klasse `java.util.Map` verarbeitet werden.

```
public interface DeploymentEvent
{
  public int getId();
  public Object getConstraint();
  public Map getArguments();
  public Object getArgument(String argumentKey);
}
```

FILE_CHANGED

Dieses Ereignis wird gefeuert, wenn Änderungen an einem Web-Objekt vorgenommen wurden. Es wird immer genutzt, wenn ein persistentes Web-Objekt verarbeitet wird, d.h. sowohl bei neuen als auch bei geänderten Objekten

Argumente

- `ARG_DEPLOYMENT_SYSTEM [String]`

  Name des Deploymentsystems

- `ARG_WEBSITE [String]`

  Name der Website

- `ARG_OID [ObjectId]`

  OID des zugehörigen VIP-Objekts

- `ARG_URL [String]`

  URL des Web-Objekts

- ARG_PATH [String]

  Pfad des Web-Objekts

- ARG_CONTENTNUMBER [Long]

  Der Wert 0 identifiziert den Standardinhalt, der Wert -1 identifiziert die dazugehörige Surrogatseite und der Wert -2 eine statifizierte Seite.

- ARG_EVENT_ID [Integer]

  Bezeichner des Ereignisses, das das Deployment ausgelöst hat (z. B. OBJECT_SUBMITTED_TO_PRODUCTION)

- ARG_OBJECT_TYPE_KEY [Long]

  Key für den Objekttyp des zugehörigen VIP-Objekts, nur für interne Zwecke bestimmt

- ARG_OBJECT_TYPE_NAME [String]

  Bezeichner für den Objekttyp des zugehörigen VIP-Objekts

FILE_DELETED

Dieses Ereignis wird gefeuert, wenn ein Web-Objekt gelöscht wurde.

Argumente

- ARG_DEPLOYMENT_SYSTEM [String]
- ARG_WEBSITE [String]
- ARG_OID [ObjectId]
- ARG_URL [String]
- ARG_PATH [String]
- ARG_CONTENTNUMBER [Long]

- ARG_EVENT_ID [Integer]
- ARG_OBJECT_TYPE_KEY [Long]
- ARG_OBJECT_TYPE_NAME [String]

FILE_URL_CHANGED

Dieses Ereignis ist nur für Standard-Deploymentsysteme verfügbar. Es wird gefeuert, wenn sich die URL eines Web-Objekts in einem Standard-Deploymentsystem ändert.

Argumente

- ARG_DEPLOYMENT_SYSTEM [String]
- ARG_WEBSITE [String]
- ARG_OID [ObjectId]

DEPLOYMENT_SYSTEM_CREATED

Dieses Ereignis wird gefeuert, wenn ein neues Deploymentsystem angelegt wird.

Argumente

- ARG_DEPLOYMENT_SYSTEM [String]
- ARG_WEBSITE [String]

DEPLOYMENT_SYSTEM_DELETED

Dieses Ereignis wird gefeuert, wenn ein Deploymentsystem gelöscht wird.

Argumente

- ARG_DEPLOYMENT_SYSTEM [String]
- ARG_WEBSITE [String]

5.2 Vorbereitungsereignisse – das Interface `PrepareEvent`

Vorbereitungsereignisse werden gefeuert, bevor die zugrunde liegende Aktion durchgeführt wird. Damit besteht die Möglichkeit, die bevorstehende Statusveränderung zu prüfen und gegebenenfalls (durch eine `VetoException`) zu verhindern. Vorbereitungsereignisse sind für Website- und System-bezogene Aktionen verfügbar, nicht für Deploymentaktionen.

Vorbereitungsereignisse werden durch Objekte des Interfaces `PrepareEvent` repräsentiert. Jedes Vorbereitungsereignis enthält als Argument eine Referenz auf das betreffende VIP-Objekt.

```
public interface PrepareEvent
{
   public int getId();
   public Object getConstraint();
   public String getUserId();
   public Map getArguments();
   public Object getArgument(String argumentKey);
```

> **Hinweis:** Vorbereitungsereignisse sind nur auf dem Master-Edit-Server verfügbar.

`PREPARE_CHECKIN_OBJECT`

Dieses Ereignis wird gefeuert, wenn ein ausgeliehenes VIP-Objekt zurückgegeben werden soll. [`checkIn`]

Argument

- `ARG_OID [ObjectId]`

PREPARE_CHECKOUT_OBJECT

Dieses Ereignis wird gefeuert, wenn ein VIP-Objekt ausgeliehen werden soll. [`checkOut`]

Argument

- `ARG_OID [ObjectId]`

PREPARE_CREATE_OBJECT

Dieses Ereignis wird gefeuert, wenn ein neues VIP-Objekt angelegt werden soll. [`create`]

Argumente

- `ARG_OID [ObjectId]`
- `ARG_NEW [RepositoryEntry]`

  Übergabe der zukünftigen Objektdaten als `RepositoryEntry`-Objekt
- `ARG_METADATA [Map]`

  Mithilfe dieses Arguments haben Sie die Möglichkeit, bereits vor dem Anlegen des Objekts die Metadaten zu ändern. So könnte z.B. die Verwendung einer bestimmten Vorlage oder eines bestimmten Objekttyps erzwungen werden.

PREPARE_DELETE_OBJECT

Dieses Ereignis wird gefeuert, wenn ein VIP-Objekt gelöscht werden soll. [`delete`]

Argument

- `ARG_OID [ObjectId]`

PREPARE_DESTROY_OBJECT

Dieses Ereignis wird gefeuert, wenn ein VIP-Objekt endgültig zerstört werden soll. [`destroy`]

Argument

- `ARG_OID [ObjectId]`

PREPARE_OBJECT_CHANGE_METADATA

Dieses Ereignis wird gefeuert, wenn die Metadaten eines VIP-Objekts geändert werden sollen. [`change`]

Argumente

- `ARG_OID [ObjectId]`
- `ARG_NEW [RepositoryEntry]`

  Übergabe der zukünftigen Objektdaten als `RepositoryEntry`-Objekt

PREPARE_REJECT_OBJECT

Dieses Ereignis wird gefeuert, wenn ein vorgelegtes VIP-Objekt abgelehnt werden soll. [`reject`]

Argument

- `ARG_OID [ObjectId]`

PREPARE_RELEASE_OBJECT

Dieses Ereignis wird gefeuert, wenn ein vorgelegtes VIP-Objekt freigegeben oder verzögert freigegeben werden soll. [`release`]

Argumente

- `ARG_OID [ObjectId]`
- `ARG_DATE [Date]`

  Datum der geplanten Freigabe

`PREPARE_RESTORE_OBJECT_VERSION`

Dieses Ereignis wird gefeuert, wenn eine frühere Version eines VIP-Objekts wiederhergestellt werden soll. [`restoreVersion`]

Argumente

- `ARG_OID [ObjectId]`
- `ARG_NEW`

  Version, die wiederhergestellt werden soll, als Objekt vom Typ `Version`

`PREPARE_SUBMIT_OBJECT_TO_QA`

Dieses Ereignis wird gefeuert, wenn ein VIP-Objekt zur Qualitätssicherung vorgelegt werden soll. [`submit`]

Argument

- `ARG_OID [ObjectId]`

`PREPARE_UNDO_CHECKOUT_OBJECT`

Dieses Ereignis wird gefeuert, wenn das Ausleihen eines Objekts rückgängig gemacht werden soll. [`undoCheckOut`]

Argument

- `ARG_OID [ObjectId]`

5.3 Ereignisbeobachter

Um auf bestimmte Ereignisse reagieren zu können, muss sich ein Agent mit einem Ereignisbeobachter auf das gewünschte Ereignis registrieren.

Wenn das Ereignis auftritt, werden alle auf dieses Ereignis registrierten Ereignisbeobachter vom Ereignisverteiler darüber informiert, indem ihre Methode `performVipEvent` bzw. `performVipDeploymentEvent` mit dem jeweiligen Ereignis als Argument aufgerufen wird.

Wenn ein Ereignisbeobachter ein Ereignis empfängt (z.B. `OBJECT_METADATA_CHANGED`), kann es sein, dass das entsprechende VIP-Objekt gar nicht mehr in der aktuellen Version existiert, z.B. weil es in der Zwischenzeit endgültig gelöscht wurde (Aktion `destroy`). Ereignisse werden asynchron verschickt. So könnte ein Workflow-Schritt bereits ausgeführt sein, bevor das Ereignis für den vorherigen Workflow-Schritt gefeuert wird. Wird das VIP-Objekt als Folge eines Ereignisses gelesen, sollte in der Methode `performVipEvent` die Version des Objekts mit der Version des Ereignisses verglichen werden. Wenn die Versionen verschieden sind oder das Objekt nicht mehr gefunden wird, kann versucht werden, das archivierte Objekt zu lesen (`ObjectHandler.get(oid,version)`). Die Version eines VIP-Objekts ist im Argument `ARG_VERSION` für jedes Ereignis abgelegt.

Ereignisbeobachter werden durch folgende Interfaces zur Verfügung gestellt:

- `EventListener` für Website- und System-bezogene Ereignisse
- `DeploymentEventListener` für Deployment-bezogene Ereignisse
- `PrepareEventListener` für Vorbereitungsereignisse

Das Interface `EventListener`

Ereignisbeobachter für Website- und System-bezogene Ereignisse müssen das Interface `EventListener` (`de.gauss.vip.api.event`) implementieren. Dieses Interface enthält eine Methode, die beim Auftreten eines registrierten Ereignisses aufgerufen wird. Diese Methode wird immer erst dann aufgerufen, wenn eine das Ereignis verursachende Aktion bereits beendet ist.

```
public interface EventListener
{
   public void performVipEvent(Event event)
           throws de.gauss.vip.api.event.WaitForMeException;
}
```

Das Interface `DeploymentEventListener`

Ereignisbeobachter für Deployment-Ereignisse müssen das Interface `DeploymentEventListener` (`de.gauss.vip.api.event`) implementieren. Dieses Interface enthält eine Methode, die beim Auftreten eines registrierten Ereignisses aufgerufen wird. Diese Methode wird immer erst dann aufgerufen, wenn eine das Ereignis verursachende Aktion bereits beendet ist.

```
public interface DeploymentEventListener
{
   public void performVipDeploymentEvent(DeploymentEvent event);
}
```

Das Interface `PrepareEventListener`

Dieser Beobachter wird benötigt, um so genannte Vorbereitungsereignisse zu verarbeiten. Für fast jede Website- und System-bezogene Aktion gibt es ein Vorbereitungsereignis (siehe Abschnitt “Vorbereitungsereignisse – das Interface `PrepareEvent`” ab Seite 121). In diesem Beobachter ist es möglich, gegen die Ausführung einer Aktion ein Veto einzulegen. Dazu muss eine `VetoException` (`de.gauss.vip.api.event`) geworfen werden.

Für das Message-Argument in der `VetoException` wird die `UserMessage`-Klasse verwendet.

```
public interface PrepareEventListener
{
   public void performVipPrepareEvent(PrepareEvent prepareEvent)
                throws VetoException;
}

public class VetoException extends Exception
{
   public VetoException(UserMessage msg);
}
```

5.4 Ereignisverteiler

Beim Auftreten eines Ereignisses verteilen zentrale Event-Dispatcher, die so genannten Ereignisverteiler, das Ereignis an alle Ereignisbeobachter, die sich für dieses Ereignis registriert haben. Dazu müssen die Ereignisbeobachter beim Ereignisverteiler registriert werden.

Zwei Ereignisverteiler stehen zur Verfügung:

1. `EventDispatcher` für die Verteilung von Website- und System-bezogenen Ereignissen
2. `DeploymentEventDispatcher` für die Verteilung von Deployment-bezogenen Ereignissen

Das Interface `EventDispatcher`

Die Registrierung eines Ereignisbeobachters erfolgt über das Interface `EventDispatcher` (diese kann über `VipRuntime.getEventDispatcher` ermittelt werden). Das Interface bietet entsprechende Methoden, um sich auf Ereignisse zu registrieren, die sich auf Websites oder das VIP-CM-System beziehen. Für Website-bezogene Ereignisse ist für das `constraint`-Argument der Methode `addListener` bzw. `removeListener` der Name einer Website zu übergeben.

Zur Registrierung eines Beobachters wird ein Ereignistyp (des Interfaces `Event` bzw. `PrepareEvent`) angegeben. Es ist dabei möglich, denselben Beobachter für mehrere Ereignistypen zu verwenden und diese dann in den entsprechenden Methoden wieder zu trennen. Die beste Lösung ist allerdings, für jeden Ereignistyp eine eigene Klasse bzw. ein eigenes Objekt anzulegen.

```
public interface EventDispatcher
{
   public void addListener(int id, EventListener obj);
   public void addListener(int id, PrepareEventListener obj);
   public void addListener(Object constraint,
                           int id, EventListener obj);
   public void addListener(Object constraint, int id,
                           PrepareEventListener obj);
   public void removeListener(int id, EventListener obj);
   public void removeListener(int id, PrepareEventListener obj);
   public void removeListener(Object constraint, int id,
                              EventListener obj);
   public void removeListener(Object constraint, int id,
                              PrepareEventListener obj);
   public void removeListener(EventListener obj);
   public void removeListener(PrepareEventListener obj);
}
```

Das Interface `DeploymentEventDispatcher`

Die Registrierung von Ereignisbeobachtern für Deployment-Ereignisse erfolgt über das Interface `DeploymentEventDispatcher`. Dieses Interface kann über `VipRuntime.getDeploymentEventDispatcher` ermittelt werden.

Das Interface bietet entsprechende Methoden, um sich auf Ereignisse zu registrieren, die sich auf das Deployment beziehen. Im `constraint`-Argument der Methode `addListener` bzw. `removeListener` wird dabei der Name des Deploymentsystems übergeben.

Zur Registrierung eines Beobachters wird ein Ereignistyp (des Interfaces `DeploymentEvent`) angegeben.

```
public interface DeploymentEventDispatcher
{
  public void addListener(int id, DeploymentEventListener obj);
  public void addListener(Object constraint,
                          int id, DeploymentEventListener obj);
  public void removeListener(int id, DeploymentEventListener
                             obj);
  public void removeListener(Object constraint, int id,
                             DeploymentEventListener obj);
  public void removeListener(DeploymentEventListener obj);
}
```

KAPITEL 6

Kontextverwaltung

Das Interface `ContextHandler` dient dazu, sich an einem VIP-CM-Server anzumelden bzw. abzumelden. Durch die Methoden dieses Interfaces sind folgende Funktionen möglich:

- Authentifizierung eines Benutzers innerhalb des VIP-CM-Systems
- Verwaltung des Kontextes einer Benutzeranmeldung
- Ändern des Passworts
- Ermittlung des zu einem Kontext gehörenden Benutzers (inklusive seines Profils mit allen direkt und indirekt zugewiesenen Websites und Funktionsbereichen)
- Möglichkeit, sich als Stellvertreter für einen anderen Benutzer anzumelden

Ein Kontext hat normalerweise nur eine begrenzte Lebensdauer. Wenn über eine bestimmte Zeit keine Aktionen vorgenommen wurden, wird der Kontext ungültig. Das VIP Java API stellt Methoden zur Verfügung, um einen Kontext trotz fehlender Aktionen über einen beliebig langen Zeitraum zu verwenden.

Neben den Möglichkeiten der Benutzeranmeldung bietet das `ContextHandler`-Interface zwei vordefinierte Kontexte, die gegebenenfalls für den Zugriff auf VIP-Objekte verwendet werden können: World und Backup (siehe Abschnitt 6.2 “Vordefinierte Kontexte” auf Seite 136).

```
public interface ContextHandler
{
   // --- Logging in and logging out of a server
   public ContextId login(String userName, String password);
   public void substituteLogin(ContextId cid, String userName);
   public void logout(ContextId cid);

   // --- Predefined logins
   public ContextId getBackupContextId();
   public ContextId getWorldContextId();

   // --- Changing user password, determining user ID
   public void changePassword(ContextId cid, char[] oldpassword,
                              char[] newpassword);
   public User getUserProfile(ContextId cid);

   // --- Extending the life of the context
   public void refreshContextId(ContextId cid);
   public long getContextTimeOut(ContextId cid);
   public void startContextRefresh(ContextId cid);
   public void startContextRefresh(ContextId cid, long refreshTime);
   public void stopContextRefresh(ContextId cid);
}
```

6.1 An- und Abmelden

Der Zugriff auf ein VIP-CM-System über das VIP Java API erfordert eine Benutzerauthentifizierung, um die Methoden der einzelnen Interfaces, insbesondere der Objektverwaltung und der Administration, überhaupt nutzen zu können. Bei vielen Aktionen werden die Funktionsbereiche des Benutzers bzw. die Zugriffssteuerungslisten der VIP-Objekte geprüft, um festzustellen, ob der Benutzer über die notwendigen Rechte verfügt.

Ein Benutzer kann sich beliebig oft anmelden. Jede Anmeldung liefert ein unterschiedliches, innerhalb eines VIP-CM-Systems eindeutiges `ContextId`-Objekt zurück.

Ein `ContextId`-Objekt versteckt die eigentliche Implementierung des Benutzerkontextes sowie die zugeordneten Informationen (welcher Benutzer sich wann und wo angemeldet hat, welche Aktionen vom Benutzer durchgeführt werden usw.). Das Interface enthält aus Sicherheitsgründen keine Methoden.

Es ist über das `ContextHandler`-Interface möglich, sich als Stellvertreter für einen anderen Benutzer anzumelden und so mit den Objektrechten dieses Benutzers zu arbeiten. Das folgende Beispiel demonstriert eine Anmeldung als Stellvertreter:

```
ContextId cid;
ContextHandler ch;
ch =VipRuntime.getContextHandler();
cid =ch.login("herbert","vip");
ch.substituteLogin(cid,"jstein ");
//---Executing actions with the cid
...
//---Log out
ch.logout(cid);
```

Nach dem Anmelden als Stellvertreter repräsentiert die vorhandene `ContextId` den Kontext des Stellvertreters und nicht mehr den Kontext des ursprünglich angemeldeten Benutzers.

Für jeden angemeldeten Benutzer kann das Profil ermittelt werden. Der folgende Aufruf liefert ein `User`-Objekt, das automatisch das Profil geladen hat.

```
ContextId cid = VipRuntime.getContextHandler().login(userName,
                                                     password);
User user = VipRuntime.getContextHandler().getUser(cid);
System.out.println("hasProfile() returns ="+user.hasProfile());
```

6.2 Vordefinierte Kontexte

Es werden von VIP ContentManager zwei vordefinierte Kontexte angeboten, die ohne Anmeldung für die Programmierung von Server-Agenten verwendet werden können:

- World – Dieser Kontext repräsentiert eine Anmeldung als Benutzer “World” (jeder Benutzer). Welche Rechte dieser Benutzer innerhalb einer Website hat, hängt von den dort definierten Zugriffssteuerungslisten ab. Die Methode `getWorldContextId` liefert diesen Kontext, der in der Regel nur bestimmte VIP-Objekte lesen kann.
- Backup – Dieser Kontext repräsentiert einen Benutzer, der lesend auf alle VIP-Objekte einer Website zugreifen darf. Dadurch kann er insbesondere für die Programmierung von Backup-Agenten eingesetzt werden. Dazu wird die Methode `getBackupContextId` verwendet.

Beide Kontexte müssen niemals erneuert werden (siehe Abschnitt 6.3 “Kontexte erneuern”) und besitzen kein Benutzerprofil.

6.3 Kontexte erneuern

Den Kontext eines angemeldeten Benutzers erhält man über die Methoden `login` bzw. `substituteLogin`. Jeder Kontext hat nur eine begrenzte Lebensdauer, falls keine Aktionen innerhalb eines gewissen Zeitintervalls durchgeführt werden. Nach einer gewissen Zeitspanne ohne Serverzugriff erlischt der Kontext. Diese Zeitspanne (*Ablaufintervall*) können Sie über das VIP-Administrationsprogramm einstellen.

Ein manuelles Erneuern des Benutzerkontextes kann mit der Methode `refreshContextId` durchgeführt werden. Nach diesem Aufruf bleibt der erneuerte Kontext für das eingestellte Ablaufintervall gültig.

Das Erneuern des Benutzerkontextes kann automatisch in einem separaten Thread durchgeführt werden. Dieser “Erneuerungs-Thread” wird mit `startContextRefresh` gestartet. Dabei kann das Zeitintervall für die Erneuerung angegeben werden. Wird es nicht angegeben, wird die Hälfte des Ablaufintervalls genommen.

Mit der Methode `stopContextRefresh` wird der zur `ContextId` gehörende Thread beendet. Insbesondere für das Durchführen von Aktionen aufgrund von empfangenen Ereignissen ist es notwendig, den Benutzerkontext automatisch erneuern zu lassen, wenn die Aktionen unter einem einheitlichen Benutzerkontext ausgeführt werden sollen. Das folgende Beispiel demonstriert dies:

```
public class DemoAgent
    implements de.gauss.vip.api.ServerAgent
{
    [...]
    private de.gauss.vip.api.admin.ContextHandler cxtHandler;
    private de.gauss.vip.api.lang.ContextId contextId;

    private final String userName;
    private final String password;
    [...]
    public DemoAgent(java.util.Properties config)
    {
        userName = config.getProperty("username");
        password = config.getProperty("password");
        [...]
    }

    public boolean start(String serverType, int majorVersion,
                         int minorVersion, String patchLevel)
    {
        try
        {
           cxtHandler = de.gauss.vip.api.VipRuntime.getContextHandler();
            contextId  = cxtHandler.login(userName, password);
            cxtHandler.startContextRefresh(contextId);
        }
        catch (de.gauss.vip.api.exception.VipApiException vax)
        {
            vax.printStackTrace();
            return false;
```

```
        }
        [...]
    }

    public void stop()
    {
        [...]
        try
        {
            cxtHandler.stopContextRefresh(contextId);
            cxtHandler.logout(contextId);
        }
        catch (de.gauss.vip.api.exception.VipApiException vax)
        { /* ignored */ }
    }
    [...]
}
```

KAPITEL 7
Objektverwaltung

Die Komponenten der Objektverwaltung sind im folgenden Diagramm zusammengestellt. Daraus wird insbesondere der Zusammenhang zwischen Objekten und Objektattributen deutlich.

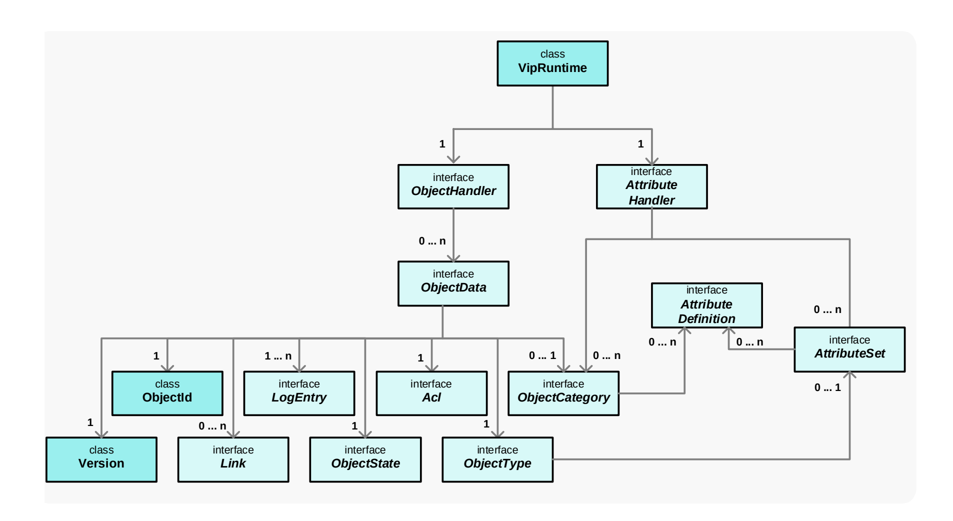


Abb. 7 – Komponenten der Objektverwaltung

Die folgenden Interfaces sind dabei von zentraler Bedeutung:

- `ObjectHandler`: Dieses Interface enthält Methoden zum Anlegen und Löschen von VIP-Objekten sowie zum Durchführen von Workflow-Aktionen (z.B. Vorlegen oder Freigeben) und Holen von Objektdaten zur Bearbeitung.

Siehe Abschnitte 7.1 “Das Interface ObjectHandler” auf Seite 142 sowie 7.2 “Aktionen und Transaktionen” auf Seite 152

- Interface ObjectData: Dieses Interface enthält Methoden zur Bearbeitung der Metadaten von VIP-Objekten.

  Siehe Abschnitt 7.3 “Objektdaten bearbeiten – das Interface ObjectData” auf Seite 173

- Interface AttributeHandler: Dieses Interface bietet Methoden zur Bearbeitung von Attributmengen und Objektkategorien.

  Siehe Abschnitt 7.5 “Attribute bearbeiten” ab Seite 185

Die Klasse ObjectHandlerUtil bietet Hilfsmethoden an, die den Zugriff auf häufig benutzte Objekte des VIP Java API vereinfachen (siehe Abschnitt 7.4 “Die Hilfsklasse ObjectHandlerUtil” auf Seite 183).

Darüber hinaus erhalten Sie in diesem Kapitel Informationen zur Suche nach Objekten über Filterfunktionen (siehe Abschnitt 7.6 “VIP-Objekte suchen” ab Seite 195) und zur Bearbeitung der Zugriffssteuerungsliste von Objekten (siehe Abschnitt 7.7 “Zugriffssteuerung” ab Seite 204).

7.1 Das Interface ObjectHandler

Der Zugriff auf die VIP-Objekte einer Website wird durch das Interface ObjectHandler ermöglicht. Um den ObjectHandler über die Methode VipRuntime.getObjectHandler anzufordern, muss als Argument der Name einer Website übergeben werden. Die Namen aller Websites, die vom Server des Agenten verwaltet werden, können mit VipRuntime.getCurrentServer.getWebsiteNames ermittelt werden. Die jeweilige Website enthält z.B. die verwendeten Objekttypen und Objektstatus. Das Website-Objekt kann über ObjectHandler.getWebsite ermittelt werden.

Ob ein VIP-Objekt bearbeitet werden kann und welche Funktionen dafür zur Verfügung stehen, hängt von mehreren Faktoren ab:

- dem Status des Objekts, siehe den folgenden Abschnitt “Objektstatus – Konstanten und mögliche Aktionen”
- der Datenhaltungssicht auf dem aktuellen VIP-CM-Server, siehe “Datenhaltungssichten” auf Seite 147
- dem Typ des Objekts, siehe “Objekttypen” auf Seite 149
- den Zugriffsrechten des angemeldeten Principals, siehe “Zugriffsrechte für VIP-Objekte” auf Seite 151

In bestimmten Fällen müssen auch Vorbedingungen für untergeordnete oder übergeordnete Objekte zutreffen. So kann ein Objekt nur freigegeben werden, wenn das übergeordnete Thema ebenfalls freigegeben ist.

Objektstatus – Konstanten und mögliche Aktionen

Jedes VIP-Objekt hat einen bestimmten Objektstatus, der den aktuellen Bearbeitungszustand beschreibt. Folgende Status gibt es:

Tabelle 6 – Konstanten der Objektstatus

Status	Konstante	Wert
geändert	`ObjectState.EDITED`	`edited`
ausgeliehen	`ObjectState.CHECKED_OUT`	`checked_out`
gelöscht	`ObjectState.DELETED`	`deleted`
abgelehnt	`ObjectState.REJECTED`	`rejected`

Status	Konstante	Wert
freigegeben	`ObjectState.RELEASED`	`released`
verzögert freigegeben	`ObjectState.PENDING_RELEASE`	`release_at`
vorgelegt	`ObjectState.SUBMITTED`	`submitted`

Der Status eines VIP-Objekts wird im VIP Java API durch das Interface `ObjectState` repräsentiert. Neben dem Namen des Status (siehe obige Konstanten) kann überprüft werden, ob bestimmte Statusübergänge möglich sind. Weiterhin kann eine lokalisierte Beschreibung des Status ermittelt werden.

Das zu einem Namen gehörende `ObjectState`-Objekt wird über eine Methode des `Website`-Objekts ermittelt (`Website.getObjectState(name)`, siehe dazu auch Abschnitt 4.7 “Informationen zu Websites” auf Seite 99).

Das Interface `ObjectState` ist folgendermaßen definiert:

```
package de.gauss.vip.api.lang;

public interface ObjectState extends Value
{
   public static final String CHECKED_OUT;
   public static final String EDITED;
   public static final String DELETED;
   public static final String REJECTED;
   public static final String RELEASED;
   public static final String PENDING_RELEASE;
   public static final String SUBMITTED;

   public String getDescription(Locale locale);
   public String getImageURL(String dplSystem);
   public String getName();
   public boolean hasTransition(ObjectState destinationState);
}
```

Abhängig vom Objektstatus sind verschiedene Aktionen, z.B. Vorlegen oder Freigeben, an einem Objekt möglich. In der Regel wird eine Aktion, wenn sie erfolgreich durchgeführt wird, zu einer Änderung des Objekt-status des jeweiligen VIP-Objekts führen. Ein zur Qualitätssicherung vorgelegtes VIP-Objekt hätte beispielsweise vor der Freigabe den Status `SUBMITTED`. Wird das VIP-Objekt durch die Methode `release` erfolgreich freigegeben, hat es danach den Status `RELEASED`.

Die folgende Tabelle bietet einen Überblick über die möglichen Aktionen für die Objektstatus und resultierende Statuswechsel.

Tabelle 7 – Objektstatus und mögliche Aktionen

Objektstatus vor der Aktion	Mögliche Aktionen	Objektstatus nach der Aktion
`EDITED`	`change`	unverändert
	`checkOut`	→ `CHECKED_OUT`
	`copy`	Quellobjekt: unverändert Zielobjekt: → `EDITED`
	`delete`	→ `DELETED`
	`directRelease`	→ `RELEASED` → `PENDING_RELEASE`
	`move`	unverändert
	`restoreVersion`	unverändert
	`submit`	→ `SUBMITTED`
`CHECKED_OUT`	`checkIn`	→ `EDITED`
	`undoCheckOut`	zurück zum Status, den das Objekt vorher hatte

Objektstatus vor der Aktion	Mögliche Aktionen	Objektstatus nach der Aktion
DELETED	destroy	Objekt wird endgültig aus dem VIP-CM-System entfernt
	reject	→ REJECTED
REJECTED	change	→ EDITED
	checkOut	→ CHECKED_OUT
	copy	Quellobjekt: unverändert Zielobjekt: → EDITED
	delete	→ DELETED
	move	unverändert
	restoreVersion	→ EDITED
	submit	→ SUBMITTED
RELEASED	change	→ EDITED
	checkOut	→ CHECKED_OUT
	copy	Quellobjekt: unverändert Zielobjekt: → EDITED
	delete	→ DELETED
	move	unverändert
	restoreVersion	→ EDITED

Objektstatus vor der Aktion	Mögliche Aktionen	Objektstatus nach der Aktion
PENDING_RELEASE	change	→ EDITED
	checkOut	→ CHECKED_OUT
	copy	Quellobjekt: unverändert Zielobjekt: → EDITED
	delete	→ DELETED
	move	unverändert
	restoreVersion	Edit-Sicht: → EDITED QS-Sicht: unverändert
SUBMITTED	reject	→ REJECTED
	release	→ RELEASED → PENDING_RELEASE

Datenhaltungssichten

Entsprechend dem Workflow von VIP ContentManager gibt es drei Sichten auf die Datenhaltung in einem VIP-CM-System: Edit, QS und Produktion. Welche Sicht auf einem VIP-CM-Server verfügbar ist, hängt vom Servertyp ab. So ist auf Produktionsservern nur die Sicht auf freigegebene VIP-Objekte verfügbar, während auf Edit-Servern alle Sichten auf die Objekte vorhanden sind. Siehe dazu auch Abschnitt “Workflow und Datenhaltungssichten” auf Seite 21.

Beim Zugriff auf VIP-Objekte muss die verwendete Datenhaltungssicht angegeben werden. Dazu dienen die Typkonstanten im Interface `Server`.

Weitere Informationen zum Interface `Server` erhalten Sie im Abschnitt 4.6 “Informationen zu VIP-CM-Servern” auf Seite 96.

Tabelle 8 – Sichtbare Status von VIP-Objekten auf verschiedenen Servern

Sicht	Typkonstante Server	Status der VIP-Objekte	Verfügbar auf Servertyp
Edit	TYPE_EDIT	alle	Edit
QS	TYPE_QA	alle außer EDITED und CHECKED_OUT	Edit, QS
Produktion	TYPE_PRODUCTION	RELEASED	Edit, QS, Produktion

Ältere Versionen der VIP-Objekte werden unabhängig von der jeweiligen Sicht in einer Versionstabelle archiviert.

Hinweis: Die verfügbaren Sichten der Deploymentsysteme auf die VIP-Objekte werden durch analoge Typkonstanten im Interface `DeploymentSystem` definiert. Siehe auch Abschnitt 4.8 “Informationen zu Deploymentsystemen” auf Seite 101.

Im Rahmen des Workflows durchläuft ein VIP-Objekt verschiedene Objektstatus und die unterschiedlichen Sichten. Die Verfügbarkeit des VIP-Objekts in den verschiedenen Datenhaltungssichten kann sich durch eine Aktion ändern: Nach der Freigabe beispielsweise ist der aktuelle Stand eines VIP-Objekts auch in der Produktionssicht verfügbar. Weitere Aktionen an dem VIP-Objekt wären in der QS-Sicht hingegen nicht mehr möglich, bis das Objekt erneut zur Qualitätssicherung vorgelegt wird.

Die Aktionen der Objektverwaltung sowie die Änderungen, die sich aus einer Aktion ergeben, sind jeweils nur in bestimmten Datenhaltungs-sichten verfügbar. In der QS-Sicht ist es z.B. nicht möglich, ein neues VIP-Objekt anzulegen, ein geändertes VIP-Objekt vorzulegen usw. Die Über-sicht im Abschnitt 7.2 “Aktionen und Transaktionen” ab Seite 152 bietet deshalb auch Angaben zu der Datenhaltungssicht, in der eine Aktion möglich ist.

Objekttypen

Jedes VIP-Objekt hat einen Objekttyp, der einerseits den verwendeten Inhalt klassifiziert (z.B. “HTML-Seite” oder “JSP-Seite”) und andererseits weitere Eigenschaften festlegt (z.B. “HTML-Vorlage” oder “Thema”) und damit die Verwendung innerhalb des VIP-CM-Systems definiert (z.B. für die Seitengenerierung). Allgemeine Informationen zu Objekttypen erhalten Sie im Abschnitt “Objekttyp” auf Seite 20.

Der Objekttyp eines VIP-Objekts wird durch das Interface `ObjectType` definiert. Neben dem eindeutigen Bezeichner eines Objekttyps sind weitere Eigenschaften spezifiziert, z.B. ob ein VIP-Objekt dieses Typs für das Anlegen zwingend eine Inhaltsdatei benötigt.

Die Zuordnung vom Namen des Objekttyps auf das zugehörige Java-Objekt wird über eine Methode des `Website`-Interfaces ermöglicht. Eine genauere Beschreibung des `Website`-Interfaces finden Sie im Abschnitt 4.7 “Informationen zu Websites” auf Seite 99. Das folgende Beispiel liefert anhand des Objekttypnamens (`HTML`) ein Objekt des Interfaces `ObjectType`.

```
ObjectHandler oh = VipRuntime.getObjectHandler("demosite");
OjectType ot = oh.getWebsite().getObjectType(ObjectType.HTML);
```

Eine lokalisierte Beschreibung des Objekttyps (z.B. für einen Client) liefert die Methode `getDescription`.

Jedem Objekttyp kann eine Attributmenge zugeordnet werden, die für diesen Objekttyp spezielle Attribute definiert. Die Methode `getAttributeSetName` liefert den Namen einer Attributmenge. Im Abschnitt 7.5 “Attribute bearbeiten” ab Seite 185 werden Attributmengen ausführlich erläutert.

```
public interface ObjectType extends Value
{
   public static final String HTML;
   public static final String TOPIC;
   public static final String TEMPLATE;
   public static final String ETC;
   public static final String GIF;
   public static final String JPG;
   public static final String PNG;
   public static final String COMPOUND;
   public static final String FRAME;
   public static final String FRAMETOPIC;
   public static final String JAVASCRIPT;
   public static final String JSP;
   public static final String JSPTOPIC;
   public static final String JSPTEMPLATE;
   public static final String FORM;
   public static final String FORMTEMPLATE;
   public static final String PDF;
   public static final String DOC;
   public static final String ASP;
   public static final String ASPTOPIC;
   public static final String PHP;
   public static final String PHPTOPIC;
   public static final String PPT;
   public static final String CGI;
   public static final String RULE;
   public static final String BOM;
   public static final String XML;
   public static final String XLS;
   public static final String XSLT;
   public static final String XMLTEMPLATE;
   public static final String XSLTTEMPLATE;

   public String getName();
   public String getAttributeSetName() throws VipApiException;
   public String getMimeType();
   public FunctionalArea getFunctionalArea();
   public boolean isTopic();
   public boolean isCompound();
   public boolean isTemplate();
   public boolean isFrame();
   public boolean isFileOnCreateNeeded();
   public boolean isSurrogateAvailable();
   public boolean isHyperLinkable();
   public boolean isMultiImportable();
   public List getCompatibleObjectTypes();
```

```
    public String getDefaultSuffix();
    public String getImageURL(String dplSystem);
    public String getDescription(Locale locale);
    public String getIndexFileName();
}
```

Zugriffsrechte für VIP-Objekte

In VIP ContentManager gibt es neun Zugriffsrechte für VIP-Objekte. Bestimmte Aktionen auf VIP-Objekte sind nur dann möglich, wenn das jeweilige Recht in der Zugriffssteuerungsliste des VIP-Objekts für den aktuellen Benutzer direkt oder indirekt vorhanden ist. Direkt zugeordnete Rechte sind die Rechte des einzelnen Benutzers. Indirekt zugeordnete Rechte sind die Rechte der Gruppe bzw. Rolle, zu der der Benutzer gehört.

Die Objektzugriffsrechte werden durch Konstanten des Interfaces `VipObjectPermission` repräsentiert, siehe Tabelle 4 “Konstanten der Zugriffsrechte für VIP-Objekte” auf Seite 86. Informationen zur Bearbeitung der Zugriffssteuerungsliste eines VIP-Objekts erhalten Sie im Abschnitt “Zugriffssteuerungslisten – das Interface `ACL`” auf Seite 206.

Ob eine Aktion an einem VIP-Objekt erlaubt ist, wird folgendermaßen überprüft: Das VIP-CM-System liest die Zugriffssteuerungsliste (ACL) des VIP-Objekts und prüft, ob das für die jeweilige Aktion erforderliche Recht für den angemeldeten Benutzer vorhanden ist. Die für Aktionen erforderlichen Rechte sind im Abschnitt “Aktionen des `ObjectHandler`-Interfaces” ab Seite 152 aufgeführt.

7.2 Aktionen und Transaktionen

Dieser Abschnitt bietet einen Überblick über die Aktionen des `ObjectHandler`-Interfaces und erläutert das Transaktionskonzept des VIP Java API.

Aktionen des `ObjectHandler`-Interfaces

Das `ObjectHandler`-Interface bietet eine Reihe von Methoden, um Aktionen an VIP-Objekten durchzuführen. In der folgenden Übersicht sind die Methoden alphabetisch sortiert. Die aufgeführten Methoden können für folgende Aufgabenstellungen genutzt werden:

Tabelle 9 – Wesentliche Methoden des `ObjectHandler`-Interfaces

Aufgabenstellung	Methoden
Objekte anlegen	"create" auf Seite 156 "multiImport" auf Seite 160
Workflow-Aktionen	"checkOut" auf Seite 155 "undoCheckOut" auf Seite 162 "checkIn" auf Seite 154 "submit" auf Seite 162 "release" auf Seite 161 "directRelease" auf Seite 158 "reject" auf Seite 161
Objekte erhalten und bearbeiten	"get" auf Seite 159 "change" auf Seite 154 "addRemark" auf Seite 154
Kopieren und Verschieben	"copy" auf Seite 156 "move" auf Seite 160

Aufgabenstellung	Methoden
Verbundobjekte konvertieren	“convertContent” auf Seite 155
Objekte suchen	“filter” auf Seite 158
Alte Version wiederherstellen	“restoreVersion” auf Seite 161
Seitengenerierung (Deployment)	“depublishPage” auf Seite 157 “generatePage” auf Seite 159 “generateTopic” auf Seite 159
Objekte löschen	“delete” auf Seite 157 “destroy” auf Seite 157

In den nachfolgenden – alphabetisch sortierten – Methodenbeschreibungen werden die Aktionen im Hinblick auf die erforderlichen Zugriffsrechte, die möglichen vorherigen Objektstatus und dem sich ergebenden Folgestatus beschrieben. Für jede Methode ist angegeben, in welcher Sicht die entsprechende Aktion verfügbar (d.h. durchführbar) ist. Vollständige, detaillierte Beschreibungen aller Methoden einschließlich der Aktionen der Objektverwaltung sind in der Online-Dokumentation (Javadoc) enthalten.

> **Hinweis:** Da auf dem Master-Edit-Server alle Datensichten zur Verfügung stehen, sind dort auch alle Aktionen erlaubt und die daraus resultierenden Objektstatus sichtbar.

addRemark

Voraussetzungen

Erforderliche Zugriffsrechte: `READ`, `WRITE_META`

Objektstatus: alle

Verfügbar in Sicht: Edit, QS

Folge

Objektstatus: ändert sich nicht

change

Voraussetzungen

Erforderliche Zugriffsrechte: `WRITE_META`, `READ` (gegebenenfalls auch `CHANGE_RIGHTS` zum Ändern der Zugriffsrechte). Für die direkte Freigabe wird der Funktionsbereich `DIRECT_RELEASE` benötigt.

Objektstatus: `EDITED`, `REJECTED`, `RELEASED`, `PENDING_RELEASE`

Verfügbar in Sicht: Edit

Folge

Objektstatus: Nach dem Ändern hat das Objekt den Status `EDITED`.

checkIn

Voraussetzungen

Erforderliche Zugriffsrechte: `WRITE`, `WRITE_META`, `READ`

Objektstatus: `CHECKED_OUT`

Verfügbar in Sicht: Edit

Folge

Objektstatus: Nach dem Zurückgeben hat das Objekt den Status `EDITED`.

`checkOut`

Voraussetzungen

Erforderliche Zugriffsrechte: `WRITE`, `WRITE_META`, `READ`

Objektstatus: `EDITED`, `REJECTED`, `RELEASED`, `PENDING_RELEASE`

Verfügbar in Sicht: Edit

Folge

Objektstatus: Nach dem Ausleihen hat das Objekt den Status `CHECKED_OUT`.

`convertContent`

Voraussetzungen

Erforderliche Zugriffsrechte: `WRITE`, `WRITE_META`, `READ`, `CREATE`, `DELETE`

Objektstatus: `EDITED`, `REJECTED`, `RELEASED`, `PENDING_RELEASE`

Verfügbar in Sicht: Edit

Folge

Objektstatus und Version des Verbundobjekts: keine

Objektstatus der neu erzeugten untergeordneten Objekte: Die untergeordneten Objekte, die während der Konvertierung erzeugt werden, erhalten den Staus `EDITED`.

Objektstatus der untergeordneten Objekte aus einer vorherigen Konvertierung: Objekte, deren Hauptversionsnummer kleiner als 1 ist, werden sofort endgültig im VIP-CM-System gelöscht [`destroy`]. Objekte, deren Hauptversionsnummer größer oder gleich 1 ist, erhalten den Status `DELETED`.

`copy`

Voraussetzungen

Je nach Objekttyp sind unterschiedliche Funktionsbereiche erforderlich (entsprechend den Zuordnungen der Objekttypen zu Funktionsbereichen im VIP-Administrationsprogramm). Für die direkte Freigabe wird der Funktionsbereich `DIRECT_RELEASE` benötigt.

Erforderliche Zugriffsrechte:

- für das kopierte VIP-Objekt: `TREE_OPERATIONS`, `WRITE_META`, `READ`
- für das Zielthema: `CREATE`, `READ`

Objektstatus: `EDITED`, `REJECTED`, `RELEASED`, `PENDING_RELEASE`

Verfügbar in Sicht: Edit

Folge

Objektstatus: Der Status des kopierten Objekts (Quellobjekt) ändert sich nicht. Die neu angelegte Kopie (einschließlich aller darin enthaltenen Unterobjekte) hat den Status `EDITED`.

`create`

Voraussetzungen

Je nach Objekttyp sind unterschiedliche Funktionsbereiche erforderlich (entsprechend den Zuordnungen der Objekttypen zu Funktionsbereichen im VIP-Administrationsprogramm). Für die direkte Freigabe wird der Funktionsbereich `DIRECT_RELEASE` benötigt.

Erforderliche Zugriffsrechte (für das übergeordnete Thema): `CREATE`, `WRITE`, `WRITE_META`, `READ`

Objektstatus: keine

Verfügbar in Sicht: Edit

Folge

Objektstatus: Nach dem Anlegen hat das Objekt den Status `EDITED`.

`delete`

Voraussetzungen

Erforderliche Zugriffsrechte: `DELETE`, `WRITE`, `WRITE_META`, `READ`

Objektstatus: `EDITED`, `REJECTED`, `RELEASED`, `PENDING_RELEASE`

Verfügbar in Sicht: Edit

Folge

Objektstatus: Nach dem Löschen hat das Objekt den Status `DELETED`.

`depublishPage`

Voraussetzungen

Erforderliche Zugriffsrechte: `RELEASE`, `READ`

Objektstatus: alle

Verfügbar in Sicht: alle

Folge

Objektstatus: ändert sich nicht

`destroy`

Voraussetzungen

Erforderliche Zugriffsrechte: `RELEASE`, `READ`

Objektstatus: `DELETED`

Verfügbar in Sicht: Edit, QS

Folge

Objektstatus: Nach dem Zerstören ist das Objekt im VIP-CM-System nicht mehr sichtbar.

`directRelease`

Voraussetzungen

Funktionsbereich `DIRECT_RELEASE`

Erforderliche Zugriffsrechte: `WRITE`, `WRITE_META`, `READ`, `RELEASE`

Im VIP-Objekt muss das Metadatum *direkte Freigabe* gesetzt sein.

Objektstatus: `EDITED`

Verfügbar in Sicht: Edit

Folge

Objektstatus: Nach der direkten Freigabe hat das Objekt vorerst den Status `SUBMITTED` und wird dann automatisch in den Status `RELEASED` bzw. `PENDING_RELEASE` überführt.

`filter`

Voraussetzungen

Erforderliches Zugriffsrecht: `READ`

Objektstatus: alle

Verfügbar in Sicht: alle

Folge

Objektstatus: ändert sich nicht

`generatePage`

Voraussetzungen

Erforderliches Zugriffsrecht: `READ`

Objektstatus: alle

Verfügbar in Sicht: alle

Folge

Objektstatus: ändert sich nicht

`generateTopic`

Voraussetzungen

Erforderliches Zugriffsrecht: `READ`

Objektstatus: alle

Verfügbar in Sicht: alle

Folge

Objektstatus: ändert sich nicht

`get`

Voraussetzungen

Erforderliche Zugriffsrechte: `READ` bzw. `READ_PRODUCTION`

Objektstatus: alle

Verfügbar in Sicht: alle

Folge

Objektstatus: ändert sich nicht

`move`

Voraussetzungen

Erforderliche Zugriffsrechte:

- für das verschobene VIP-Objekt: `TREE_OPERATIONS`, `WRITE_META`, `READ`
- für das Zielthema: `CREATE`, `READ`

Objektstatus: `EDITED`, `RELEASED`, `PENDING_RELEASE`, `REJECTED`

Verfügbar in Sicht: Edit

Folge

Objektstatus: Nach dem Verschieben hat das verschobene Objekt selbst den Status `EDITED`. Der Status der untergeordneten Objekte bleibt unverändert.

`multiImport`

Voraussetzungen

Der Funktionsbereich `IMPORT` ist erforderlich.

Erforderliche Zugriffsrechte (für das übergeordnete Thema): `CREATE`, `WRITE`, `WRITE_META`, `READ`

Objektstatus: alle

Verfügbar in Sicht: Edit

Folgen

Objektstatus: Nach dem Multiimport haben die angelegten Objekte den Status `EDITED`.

`reject`

Voraussetzungen

Erforderliche Zugriffsrechte: `RELEASE`, `READ`

Objektstatus: `SUBMITTED`, `DELETED`

Verfügbar in Sicht: Edit, QS

Folge

Objektstatus: Nach dem Ablehnen hat das Objekt den Status `REJECTED`.

`release`

Voraussetzungen

Erforderliche Zugriffsrechte: `RELEASE`, `READ`

Objektstatus: `SUBMITTED`

Verfügbar in Sicht: Edit, QS

Folge

Objektstatus: Nach der Freigabe hat das Objekt den Status `RELEASED` bzw. `PENDING_RELEASE`, wenn das Metadatum “Verzögerte Freigabe” gesetzt ist.

`restoreVersion`

Voraussetzungen

Erforderliche Zugriffsrechte: `WRITE`, `WRITE_META`, `READ`

Objektstatus: `EDITED`, `REJECTED`, `RELEASED`, `PENDING_RELEASE`

Verfügbar in Sicht: Edit

Folge

Objektstatus: Nach dem Wiederherstellen der Version hat das Objekt den Status `EDITED`.

`submit`

Voraussetzungen

Erforderliche Zugriffsrechte: `WRITE`, `WRITE_META`, `READ`

Objektstatus: `EDITED,REJECTED`

Verfügbar in Sicht: Edit

Folge

Objektstatus: Nach dem Vorlegen hat das Objekt den Status `SUBMITTED`.

`undoCheckOut`

Voraussetzungen

Erforderliche Zugriffsrechte: `WRITE`, `WRITE_META`, `READ`

Objektstatus: `CHECKED_OUT`

Verfügbar in Sicht: Edit

Folge

Objektstatus: Nachdem das Ausleihen des Objekts rückgängig gemacht wurde, hat das Objekt den Status, den es vor dem Ausleihen hatte.

Synchrone und asynchrone Methoden

Die Methoden des Interfaces `ObjectHandler` sind in synchrone und asynchrone Methoden unterteilt. Asynchrone Methoden erkennen Sie daran, dass diese mit dem Suffix `-Async` enden. Bei allen anderen Methoden handelt es sich um synchrone Methoden.

In der folgenden Tabelle sind die Methoden des `ObjectHandler`-Interfaces zusammengestellt, die synchron oder asynchron aufgerufen werden können.

Tabelle 10 – Synchrone und asynchrone Methoden

Synchrone Methode	Asynchrone Methode
change	changeAsync
copy	copyAsync
delete	deleteAsync
depublishPage	depublishPageAsync
destroy	destroyAsync
directRelease	directReleaseAsync
generatePage	generatePageAsync
move	moveAsync
multiImport	multiImportAsync
reject	rejectAsync
release	releaseAsync
submit	submitAsync
undoCheckOut	undoCheckOutAsync

Der Unterschied zwischen den synchronen und den asynchronen Methoden liegt im Zeitpunkt ihrer Rückkehr. Ruft man eine synchrone Methode auf, dann kehrt diese zurück, wenn die eigentliche Aufgabe abgeschlossen und das lokale Deployment informiert wurde (das Deployment wurde zu diesem Zeitpunkt aber noch nicht durchgeführt). Im Gegensatz dazu kehrt die asynchrone Methode praktisch sofort nach ihrem Aufruf wieder zurück. Man kann also schon die nächste VIP Java API-Methode aufrufen, obwohl die Aktionen, die zur Durchführung der ersten Methode notwendig sind, noch im Hintergrund ablaufen.

Zur Verdeutlichung nehmen wir an, dass Sie einen relativ großen Teilbaum in ein anderes Thema kopieren möchten. Anschließend möchten Sie an einer anderen Stelle der Website diverse andere Aktionen ausführen. Nach einem synchronen Aufruf der `copy`-Methode werden die nachfolgenden Methoden erst ausgeführt, wenn der Server mit dem Kopieren aller im Teilbaum enthaltenen Objekte fertig ist (abgesehen vom Deployment). Ruft man hingegen die entsprechende asynchrone Methode `copyAsync` auf, dann werden die Kopieraktionen auf dem Server angestoßen, aber die Methode kehrt sofort zurück und die nachfolgenden Methoden werden ausgeführt.

Die Verwendung von asynchronen Methoden ist besonders sinnvoll, wenn Sie Aktionen anstoßen, deren Bearbeitung längere Zeit dauern kann, Sie aber nicht auf den Abschluss dieser Aktionen warten möchten. Da die asynchrone Methode sofort nach dem Anstoß der Aktion auf dem Server zurückkehrt, haben Sie die Möglichkeit, in der Zwischenzeit andere Aktionen an *hiervon unabhängigen Objekten* durchzuführen. Tritt bei einer asynchronen Aktion ein Fehler auf, dann können Sie die gesamte Aktion wieder rückgängig machen. Bei synchronen Methoden ist das nicht unbedingt möglich.

Transaktionen

Mehrere Aktionen an VIP-Objekten können innerhalb einer Transaktion ausgeführt werden, d.h., die Änderungen werden erst wirksam, wenn die Transaktion explizit beendet wird.

Eine Transaktion wird über die Methode `beginTransaction` des `ObjectHandler`-Interfaces angefordert. Es wird ein Transaktionskontext (ein Objekt des Interfaces `ContextId`) zurückgeliefert, der als Argument für weitere Aktionen (z.B. `create`, `submit` oder `release`) verwendet werden kann. Alle Aktionen, die diesen Transaktionskontext verwenden, werden innerhalb der Transaktion ausgeführt und erst wirksam, wenn die Methode `commitTransaction` aufgerufen wird. Die Änderungen werden nicht durchgeführt, wenn `rollbackTransaction` aufgerufen wird.

Wichtig: Eine angeforderte Transaktion muss grundsätzlich abgeschlossen und dadurch für das VIP-CM-System wieder freigegeben werden, d.h., es muss immer entweder `rollbackTransaction` oder `commitTransaction` aufgerufen werden.

Mithilfe der Funktion `beginTransaction` ist es also möglich, mehrere Methodenaufrufe zu einer einzigen Aktion zu bündeln. Mit `rollbackTransaction` wird die laufende Transaktion abgebrochen bzw. werden die durchgeführten Änderungen rückgängig gemacht. Bei erfolgreicher Durchführung werden entsprechende Änderungen durch die Methode `commitTransaction` bestätigt und dadurch permanent gemacht. Dies bedeutet, dass alle in einer Transaktion gebündelten Aktionen entweder insgesamt durchgeführt oder rückgängig gemacht werden.

Hinweise

- Transaktionen können nur auf dem Master-Edit-Server angefordert und abgeschlossen werden, d.h., die Methoden `beginTransaction` und `commitTransaction` sind nur dort verfügbar.
- Werden Aktionen innerhalb einer Transaktion ausgeführt, dann werden die zugehörigen Events erst nach dem erfolgreichen Ende der Transaktion gefeuert.
- Der Fortschritt der Transaktionsbearbeitung kann über das `Progress`-Objekt festgestellt werden (siehe Abschnitt “Transaktionskontrolle – das Interface `Progress`” auf Seite 167).

Kontexte (Benutzer- und Transaktionskontext)

Jede Aktion wird innerhalb eines Kontextes ausgeführt. Dabei wird zwischen einem Benutzerkontext und einem Transaktionskontext unterschieden. Beide Kontexte werden durch Objekte des Interfaces `ContextId` repräsentiert.

Einen Benutzerkontext erhält man als Ergebnis des Anmeldevorgangs. Er enthält Informationen über den angemeldeten Benutzer, insbesondere die Benutzerkennung und eine Angabe darüber, wann die letzte Aktion von diesem Benutzer ausgeführt wurde. Ein Benutzerkontext wird ungültig, wenn während einer bestimmten Zeitspanne keine Aktion ausgeführt wurde.

Ein Transaktionskontext erweitert den Benutzerkontext. Er enthält eine Referenz auf die entsprechende Transaktion. Während einer Transaktion ist es möglich, den Fortschritt der Aktion zu erfragen und diese gegebenenfalls vorzeitig abzubrechen. Der Transaktionskontext wird ungültig, sobald die Aktion beendet wird. Wenn sie erfolgreich war, wird erst danach der Systemzustand aktualisiert.

Hinweise:

Wenn eine synchrone oder eine asynchrone Aktion beendet wurde, d. h. die Transaktion erfolgreich abgeschlossen wurde, werden die Objektdaten auf allen Servern mit dem Datenbestand des Master-Edit-Servers aktualisiert. Allerdings wird nicht auf die betroffenen Deploymentsysteme der Server gewartet. Die URL für ein Deploymentsystem referenziert daher unter Umständen eine Ressource, die noch nicht vom Deploymentsystem generiert wurde.

Das VIP-CM-System sperrt Objekte, die in einer laufenden Transaktion bearbeitet werden. Diese Objekte stehen deshalb für eventuell parallel laufende Aktionen nicht zur Verfügung.

Transaktionskontrolle – das Interface `Progress`

Für jeden Transaktionskontext kann der Bearbeitungsfortschritt durch ein Objekt des Interfaces `Progress` ermittelt werden. Diese Fortschrittskontrolle kann durch eine erneute Anfrage über das API (`Progress.update`) aktualisiert werden.

Ein `Progress`-Objekt kann folgende Zustände haben:

- `RUNNING`: Die Transaktion ist gerade in Bearbeitung.
- `FINISHED`: Die Transaktion wurde erfolgreich beendet.
- `DEPLOY`: Ein lokal installiertes Deploymentsystem hat noch Aufträge für diese Transaktion.
- `DEPLOY_FINISHED`: Die Deploymentaufträge für diese Transaktion (auf den lokal installierten Deploymentsystemen) sind abgeschlossen.

- `ERROR`: Die Transaktion wurde aufgrund eines Fehlers oder durch eine explizite Abbruchaufforderung (`rollbackTransaction`) beendet.
- `INVALID`: Es wurde versucht, den Fortschritt einer bereits beendeten Transaktion zu ermitteln. Der Transaktionskontext ist ungültig.

Wenn der aktuelle Server Deploymentsysteme besitzt, wechselt der Zustand des `Progress`-Objekts von `FINISHED` auf `DEPLOY`. In diesem Fall kann gewartet werden, bis alle lokal installierten Deploymentsysteme die zur Transaktion gehörenden Deploymentaufträge abgearbeitet haben (alle Seiten wurden generiert), oder es kann die aktuelle Anzahl der Aufträge sowie der bisher bearbeiteten Aufträge ausgelesen werden.

```
package de.gauss.vip.api.object.Progress

public interface Progress
{
   public final int RUNNING;
   public final int FINISHED;
   public final int ERROR;
   public final int INVALID;
   public final int DEPLOY;
   public final int DEPLOY_FINISHED;

   public String getMessage (Locale locale);
   public Date getStartDate ();
   public int getNumberOfSteps ();
   public int getCurrentStep ();
   public int getState ();
   public VipApiException getException ();
   public Value getResult ();
   public void update ();
   public boolean waitForDeployment (String dplSystem,long timeout);
   public int getRemainingJobs (String dplSystem);
   public int getProcessedJobs (String dplSystem);
   public int getNumberOfFailedSteps();
   public int getNumberOfExecutedSteps();
   public Long getTransactionId();

}
```

Für den Fortschritt einer Transaktion ergeben sich folgende mögliche Verläufe:

- INVALID
- RUNNING → ERROR
- RUNNING → FINISHED
- RUNNING → FINISHED → DEPLOY → DEPLOY_FINISHED

Hinweise:

Der Fortschritt einer Transaktion kann nur so lange ermittelt werden, wie der Transaktionskontext gültig bleibt. Nach dem Ende einer Transaktion ist der Transaktionskontext ungültig (unabhängig davon, ob die Transaktion erfolgreich war oder nicht).

Der Fortschritt des Deployments kann nur für lokal installierte Deploymentsysteme ermittelt werden.

Beispiel

Die folgenden VIP-Objekte sollen synchron vorgelegt und asynchron freigegeben werden (OIDs): 1, 3, 10, 15 (Ausführung auf dem Master-Edit-Server). Für die Freigabe-Aktion wird der Fortschritt angezeigt. Danach wird so lange gewartet, bis alle lokal installierten Deploymentsysteme die zur Transaktion gehörenden Aufträge beendet haben.

```
ObjectHandler oh = VipRuntime.getObjectHandler("website");
ContextId cid = VipRuntime.getContextHandler().login(userid,password);
List l = ObjectHandlerUtil.toOIDList("1,3,10,15");
oh.submit(cid,l,null,null,null);
ContextId tid = oh.releaseAsync(cid,l,null,null);
Progress prog = oh.getProgess(tid);
while (prog.getState()==Progress.RUNNING)
{
   System.out.println("state="+prog.getMessage(Locale.getDefault())+
      " step="+prog.getCurrentStep()+" max="+prog.getNumberOfSteps());
   prog.update();
   Thread.currentThread().sleep(200);
}
if (prog.getState()==Progress.ERROR)
```

```
{
   System.out.println("error ="+prog.getException().getMessage());
}
else // FINISHED
{
   prog.update();
   while (prog.getState()==Progress.DEPLOY)
   {
         Thread.currentThread().sleep(200);
         System.out.println("Local deployment active...");
         prog.update();
   }
}
```

Aktionen und Deployment

Das eigentliche Durchführen einer Aktion dauert an, bis die entsprechenden Änderungen an den VIP-Objekten auf dem Master-Edit-Server persistent gespeichert und alle Proxy-Server über die Änderungen benachrichtigt worden sind. Der Abschluss einer Aktion bedeutet aber nicht unbedingt, dass für die entsprechenden VIP-Objekte von den Deploymentsystemen bereits Web-Objekte generiert wurden, die die vorgenommenen Änderungen widerspiegeln. Die Aktion und das Deployment sind voneinander entkoppelt. Das muss bei einem Zugriff auf die generierten Seiten (nach einer Aktion) berücksichtigt werden. Die relevanten Deploymentaufträge sind womöglich noch nicht durchgeführt worden, sodass unter Umständen noch keine URL für ein Web-Objekt auf einem bestimmten Deploymentsystem verfügbar ist.

Angaben zum Verlauf des Deployments können über die folgenden Methoden ermittelt werden:

- `Progress.getProcessedJobs` – Gibt die Anzahl der bereits fertig gestellten Deploymentaufträge (für eine bestimmte Transaktion) zurück.
- `Progress.getRemainingJobs` – Gibt die Anzahl der noch ausstehenden Deploymentaufträge (für eine bestimmte Transaktion) zurück.

Es besteht außerdem die Möglichkeit, auf den Abschluss der mit einer Aktion verbundenen Deploymentaufträge (auf einem bestimmten Deploymentsystem) zu warten.

Dazu kann bei synchronen Workflow-Aktionen die entsprechende Methode des `ObjectHandler`-Interfaces mit einem zusätzlichen Parameter (`DeploymentWaitInfo`) aufgerufen werden. Mit dem Parameter kann auch festgelegt werden, wie lange auf den Abschluss des Deployments gewartet werden soll. Das folgende Beispiel legt ein neues VIP-Objekt an und wartet maximal 10 Sekunden auf das angegebene Deploymentsystem.

```
ObjectHandler oh = ...
ObjectData objectData = oh.createInitialObjectData(parentOID,
templateOID,
   type, title);
DeploymentWaitInfo dwi = new DeploymentWaitInfo("demowebsite_edit",
   10000);
ObjectId oid = oh.create(cid, objectData, content, remark, dwi);
if (!dwi.isTimeout())
{
    ObjectData od = oh.get(cid, oid);
    URL url = od.getURL("demowebsite_edit");
    ...
}
```

Bei asynchronen Workflow-Aktionen kann ebenfalls auf den Abschluss des Deployments gewartet werden. Dazu dient die Methode `waitForDeployment` des `Progress`-Interfaces. Im folgenden Beispiel wird ein Multiimport asynchron durchgeführt und auf das angegebene Deploymentsystem bis zu 10 Sekunden gewartet.

```
ContextId tid = oh.multiImportAsync(cid, parentOid, title,
   startFile, initialData, remark);
Progress prog = oh.getProgress(tid);
if (prog.waitForDeployment("demowebsite_edit", 10000))
{
   ... // deployment finished
}
```

Hinweis: Für das Warten auf den Abschluss relevanter Deploymentaufträge ist eine bidirektionale Verbindung mit dem entsprechenden Deploymentsystem notwendig: Die Kommunikation mit dem Deploymentsystem darf in diesem Fall nicht durch eine Firewall (einseitig) unterbunden sein.

7.3 Objektdaten bearbeiten – das Interface `ObjectData`

Mithilfe der Methoden des VIP Java API können Sie die einzelnen Bestandteile von VIP-Objekten bearbeiten. Dazu stellt das Interface `ObjectData` entsprechende Funktionen zur Verfügung:

- Metadaten von VIP-Objekten lesen und ändern (siehe folgenden Abschnitt)
- OID und Version von VIP-Objekten abfragen (siehe Abschnitt “OID und Version von VIP-Objekten lesen” auf Seite 178)
- Referenzen von VIP-Objekten bearbeiten (siehe Abschnitt “Referenzen bearbeiten” auf Seite 180)
- Protokoll auslesen (siehe Abschnitt “Protokoll lesen” auf Seite 181)
- Inhalt bearbeiten (siehe Abschnitt “Inhalt bearbeiten” auf Seite 182)

Standardmetadaten lesen und ändern

Die folgende Tabelle enthält die Metadaten, die für jedes VIP-Objekt definiert sind. Die zugehörigen Methoden zum Lesen bzw. zum Ändern dieser Werte sind im Interface `ObjectData` (`de.gauss.vip.api.object`) definiert. Die Objektdaten des aktuellen bzw. eines versionierten VIP-Objekts werden über das `ObjectHandler`-Interface ermittelt (Methode `get`).

Weitere Hinweise zum Ändern von Metadaten enthält Abschnitt 7.5 “Attribute bearbeiten” auf Seite 185.

Tabelle 11 – Methoden für den Zugriff auf Standardmetadaten

Eigenschaft	Methode[a]	Datentyp	get	set	Null-werte
OID	`ObjectId`	`ObjectId`	✓	✗	✗
Vorlage	`Template`	`ObjectId`	✓	✓	✓
Titel	`Title`	`String`	✓	✓	✗
Überschrift	`Subtitle`	`String`	✓	✓	✓
Zugriffs-steuerungsliste[b]	`ACL`	`Acl`	✓	✓	✓
Untergeordnete Objekte in der Navigations-sicht	`Children`	`List [ObjectId]`	✓	✗	✗
Erstellungs-datum	`CreatedDate`	`Date`	✓	✗	✗
Änderungs-datum	`ModifiedDate`	`Date`	✓	✗	✓
Freigabedatum	`ReleasedDate`	`Date`	✓	✗	✓
Freigeben am	`Pending ReleaseDate`	`Date`	✓	✓	✓
Typ	`ObjectType`	`ObjectType`	✓	✓	✗
Status	`ObjectState`	`ObjectState`	✓	✗	✗
Objektkategorie	`ObjectCategory`	`ObjectCategory`	✓	✓	✓
Ablaufdatum	`ExpireDate`	`Date`	✓	✓	✓
Autor	`CreatedBy`	`User`	✓	✗	✗
Geändert von	`ModifiedBy`	`User`	✓	✗	✓

Eigenschaft	Methode[a]	Datentyp	get	set	Null-werte
Freigegeben von	`ReleasedBy`	`User`	✓	✗	✓
Vorschlag für Dateiname	`DeploymentHint`	`String`	✓	✓	✗
Beschreibung	`Description`	`String`	✓	✓	✓
E-Mail Edit[c]	`EditEMail Receiver`	`Set [String]`	✓	✓	✓
E-Mail QS[d]	`QAEMail Receiver`	`Set [String]`	✓	✓	✓
E-Mail Freigabe[e]	`ReleaseEMail Receiver`	`Set [String]`	✓	✓	✓
Version	`Version`	`Version`	✓	✗	✗
Direkte Freigabe[f]	`DirectRelease`	`boolean`	✓	✓	✗
Sprache	`Locale`	`Locale`	✓	✓	✓
Schlagwörter	`Keywords`	`Set [String]`	✓	✓	✓
Zielgruppe	`TargetGroup`	`String`	✓	✓	✓
Übergeord-netes Thema[g]	`Topic`	`ObjectId`	✓	✗	✓
Referenzen vom Inhalt anderer Doku-mente auf dieses Dokument	`LinkedFrom`	`Set [ObjectId]`	✓	✗	✗

Eigenschaft	Methode[a]	Datentyp	get	set	Null-werte
Referenzen des Inhalts auf andere Dokumente	`LinksTo`	`Set [Link]`	✓	✓	✗
Name der Website	`WebsiteName`	`String`	✓	✗	✗

(a) Der tatsächliche Methodenname wird durch das Präfix `get` bzw. `set` ergänzt, also z.B. `getACL` oder `setACL`.

(b) Zugriffsrecht “Rechte ändern” erforderlich.

(c) Werden benachrichtigt, wenn das VIP-Objekt abgelehnt wurde oder es abgelaufen ist.

(d) Werden benachrichtigt, wenn das VIP-Objekt vorgelegt wurde.

(e) Werden benachrichtigt, wenn das VIP-Objekt freigegeben wurde.

(f) Funktionsbereich “Direkte Freigabe” erforderlich.

(g) Das übergeordnete Thema kann mit der Methode `ObjectHandler.move` geändert werden.

Folgende Metadaten werden vom Deploymentsystem berechnet und erwarten als Argument immer den Namen eines Deploymentsystems.

Tabelle 12 – Vom Deploymentsystem abhängige Metadaten

Eigenschaft	Methode	Datentyp	get	set	Null-werte
URL der generierten Seite	`URL`	`URL`	✓	✗	✓
Absoluter Dateipfad der generierten Seite	`Pathname`	`String`	✓	✗	✓
URL einer generierten Hilfsseite (z.B. für Bilder)	`SurrogateURL`	`URL`	✓	✗	✓

Hinweise zum Ändern von Attributwerten

Für das Ändern von Attributwerten über das `ObjectData`-Interface sind einige Besonderheiten zu beachten.

Die Objektdaten eines VIP-Objekts werden über die entsprechenden `set`-Methoden gesetzt. Sie werden allerdings erst wirksam, wenn die Aktion `change` des `ObjectHandler`-Interfaces aufgerufen wird. Es kann vorkommen, dass ein anderer Benutzer ebenfalls die Metadaten desselben VIP-Objekts geändert hat und seine Änderung über die Methode `ObjectHandler.change` wirksam geworden ist. Die aktuellen Objektdaten sind dadurch ungültig geworden. Dies wird signalisiert, indem die Methode `isInvalid` (des `ObjectData`-Interfaces) `true` zurückliefert. Ein Aufruf von `ObjectHandler.change` mit ungültigen Metadaten resultiert in einer `VipApiException`. Das folgende Code-Beispiel minimiert diese Fehlermöglichkeit.

```
ObjectData od = ...
od.setTitle("Neuer Title");
od.setTemplate(null);
if (!od.isInvalid())
{
   VipRuntime.getObjectHandler("demosite").change(cid, od, "remark");
}
```

Einige Attribute erlauben es, dass sie auf den Wert `null` gesetzt werden (z.B. `setSubtitle(null)`). Für die Zugriffssteuerungsliste (ACL) hat dies eine besondere Semantik: `setACL(null)` bewirkt, dass die eventuell vorhandenen eigenen Zugriffssteuerungsliste nicht mehr verwendet werden soll. Es wird dann die Zugriffssteuerungsliste des in der Navigationshierarchie übergeordneten VIP-Objekts verwendet, das eine eigene Zugriffssteuerungsliste besitzt. Im Zweifelsfall besitzt immer das Wurzelobjekt der Website eine eigene Zugriffssteuerungsliste. Dadurch kann der Vererbungsmechanismus der Zugriffssteuerungsliste explizit wieder eingeschaltet werden.

OID und Version von VIP-Objekten lesen

Eine Referenz auf ein VIP-Objekt wird durch ein Objekt der Klasse `ObjectId` repräsentiert.

```
public class ObjectId implements Key, Value
{
   public ObjectId(String oid)
   public ObjectId(Key oid)
   public String getId()
   public StringValue getIdAsValue()
   public int hashCode()
   public boolean equals(Object obj)
   public int compareTo(Object obj)throws ClassCastException
   public Object clone()
   public String format(Locale locale)
   public boolean isNull()
   public Object getValue()
   public String getKey()
   public String toString()
}
```

Jedes verwaltete VIP-Objekt besitzt eine Version, die durch ein Objekt der Klasse `Version` (`de.gauss.vip.api.lang`) repräsentiert wird. Eine Version besteht aus drei Nummern für Hauptversion (Major), Nebenversion (Minor) und Mikroversion (Micro). Die Mikroversion ändert sich, wenn ein VIP-Objekt in der Edit-Sicht geändert wird (Ausleihen, Metadaten ändern). Die Nebenversion ändert sich, wenn ein VIP-Objekt zur Qualitätssicherung vorgelegt wird, wobei dann die Mikroversion auf 0 gesetzt wird. Die Hauptversion wird beim Freigeben erhöht, wobei die Nebenversion auf 0 gesetzt wird.

```
public class Version implements Value
{
   public Version(int major, int minor, int micro)
public int getMajor()
public int getMinor()
public int getMicro()
public String toString()
public int hashCode()
public int compareTo(Object another) throws ClassCastException
public boolean equals(Object obj)
public Object clone()
public String format(Locale locale)
public boolean isNull()
public Object getValue()
public String getKey()
}
```

Referenzen bearbeiten

Die Methode `setLinksTo` des `ObjectData`-Interfaces erlaubt es, Referenzen zum Inhalt des VIP-Objekts hinzuzufügen bzw. zu entfernen. Eine Referenz kann dabei ein vom VIP-CM-System verwaltetes VIP-Objekt oder eine externe URL auf ein beliebiges Dokument sein.

Referenzen werden durch Objekte des Interfaces `Link` (`de.gauss.vip.api.lang`) repräsentiert. Eine konkrete Klasse, die dieses Interface implementiert und die für die Methode `setLinksTo` verwendet werden muss, ist die Klasse `SimpleLink` (`de.gauss.vip.api.object`).

```
public interface Link extends Value
{
   public ObjectId getOID();
   public String getURL();
   public int hashCode();
   public boolean isExternal();
   public boolean isFrameLink();
   public boolean isInContent();
}

public class SimpleLink implements Link
{
   public SimpleLink(ObjectId oid)
   public SimpleLink(String url)
   public ObjectId getOID();
   public String getURL();
   public boolean isExternal();
   public boolean isFrameLink();
   public boolean isInContent();

   public int hashCode():
   public boolean equals(Object obj);
   public int compareTo(Object obj) throws ClassCastException;
   public Object clone();
   public String format(Locale locale);
   public boolean isNull();
   public Object getValue();
   public String toString();
}
```

Protokoll lesen

Die Historie eines VIP-Objekts wird mit der Methode `getLastLogEntries` des `ObjectData`-Interfaces ermittelt. Das erste Argument dieser Methode gibt die Nummer des Protokolleintrags an, das zweite bestimmt, wie viele Eintrage ermittelt werden sollen. Dabei wird immer rückwärts gezählt.

```
public interface LogEntry extends Serializable
{
   public int getNumber();
   public Date getDate();
   public User getUser();
   public String getMessage(Locale locale);
   public String getRemark();
   public String getVersion();
}
```

Alle Einträge können wie folgt bestimmt werden:

```
ContextId cid = VipRuntime.getContextHandler().login(userName,
                                                      password);
ObjectData od = VipRuntime.getObjectHandler("demosite").get(cid,
                                             new ObjectId("100"));
List l = od.getLastLogEntries(0, -1)
```

Das folgende Beispiel bestimmt die letzten zehn Protokolleinträge.

```
int last = od.getNoOfLogEntries();
Iterator i = od.getLastLogEntries(last-1, 10).iterator();
while (i.hasNext())
{
   LogEntry e = (LogEntry)i.hasNext();
   System.out.println("Log:"+e.getDate()+" "+e.getUser().getName
      +" "+e.getMessage(Locale.ENGLISH));
}
```

Inhalt bearbeiten

Der Inhalt eines VIP-Objekts wird von VIP ContentManager in einer Datenbanktabelle gespeichert. Der dort abgelegte Inhalt kann über die Methode `getContent` des `ObjectData`-Interfaces ausgelesen werden. Der Inhalt wird dabei in das angegebene `OutputStream`-Objekt kopiert. Weiterhin kann geprüft werden, ob ein VIP-Objekt überhaupt einen Inhalt hat (`hasContent`) und, wenn ja, aus wie vielen Bytes der Inhalt besteht (`getContentSize`).

Das folgende Beispiel liest den Inhalt eines VIP-Objekts in die Datei **test.tmp**.

```
ObjectData od = ...
if (od.hasContent())
{
   FileOutputStream out = new FileOutputStream("test.tmp");
   od.getContent(out);
   out.close();
}
```

Die exakte Signatur der genannten Methoden lautet wie folgt:

```
public boolean hasContent();
public int getContentSize();
public void getContent(OutputStream out);
public File getCheckOutContent(ContextId cid, String dplSystem,
   File destinationFile);
```

Zwischen `getContent` und `getCheckOutContent` besteht folgender Unterschied: `getCheckOutContent` passt die im Inhalt verwendeten Verweise an das Deploymentsystem an, das als Argument angegeben ist. Diese Methode liefert also den Inhalt, der z.B. im CMS-Client für die Bearbeitung verwendet wird.

7.4 Die Hilfsklasse `ObjectHandlerUtil`

Die Klasse `ObjectHandlerUtil` bietet Hilfsmethoden an, die den Zugriff auf häufig benutzte Objekte des VIP Java API vereinfachen. Die Hilfsmethoden dieser Klasse unterstützen Sie bei folgenden Aufgaben:

- Ermitteln der Objekttypen einer Website: Es gibt eine Reihe von Methoden im `ObjectHandler`-Interface, die als Argument einen Objekttyp benötigen. Im VIP-CM-System sind die Objekttypen jeweils für eine Website definiert und daher über das Interface `Website` erreichbar. Der Zugriff auf diese Information über die Methoden im `ObjectHandler`-Interface bzw. im `Website`-Interface ist nicht ganz einfach (siehe Beispiel).
- Bestimmen der Attributmengen und Objektkategorien
- Erzeugen der Listen von OIDs: Die meisten Methoden des `ObjectHandler`-Interfaces benötigen als Argument eine Liste von OIDs. Die Methode `ObjectHandlerUtil.toOIDList()` kann verwendet werden, um aus einer Zeichenkette eine Liste von einer oder mehreren OIDs zu erzeugen.
- Ermitteln der Produktversion von VIP ContentManager

```
public final class ObjectHandlerUtil
{
   public static List toOIDList(ObjectId oid)
   public static List toOIDList(String oids)
   public static Locale[] getLocales()
   public static File getTemporaryDirectory(String websiteName)
   public static String getMailHost()
   public static ObjectType[] getObjectTypes(String websiteName)
   public static ObjectType getObjectType(String websiteName,
      String objectTypeName)
   public static ObjectState[] getObjectStates(String websiteName)
   public static ObjectState getObjectState(String websiteName,
      String objectStateName)
   public static String[] getObjectCategoryNames(String websiteName)
   public static AttributeDefinition[] getObjectCategoryAttributes
     (String websiteName, String objectCategoryName)
```

```
    public static AttributeDefinition getObjectCategoryAttribute
      (String websiteName, String objectCategoryName, Key key)
    public static String[] getAttributeSetNames(String websiteName)
    public static AttributeDefinition[] getAttributeSetAttributes
      (String websiteName, String attributeSetName)
    public static AttributeDefinition getAttributeSetAttribute
      (String websiteName, String attributeSetName, Key key)
    public static int getVIPVersionMajorNumber()
    public static int getVIPVersionMinorNumber()
    public static String getVIPVersionMajorString()
    public static String getVIPVersionMinorString()
    public static String getVIPVersionPatchlevel()
}
```

Beispiel

Das folgende Beispiel zeigt, wie der Objekttyp mit und ohne Verwendung der Hilfsklasse ermittelt werden kann.

```
ObjectType type;
// without auxiliary class
ObjectHandler oh = VipRuntime.getObjectHandler("DemoWebsite");
type = oh.getWebsite().getObjectType(ObjectType.HTML);
// or
Website w = VipRuntime.getCurrentServer().getWebsite("DemoWebsite");
type = w.getObjectType(ObjectType.HTML);
// with auxiliary class
type = ObjectHandlerUtil.getObjectType("DemoWebsite", ObjectType.HTML);
```

7.5 Attribute bearbeiten

Alle Attribute eines VIP-Objekts implementieren das Interface `Key`. Die Attributwerte implementieren das Interface `Value`. Dieses Interface erweitert – wie im folgenden Klassendiagramm dargestellt – die Interfaces `Comparable`, `Cloneable` und `Serializable` des Java SDK.

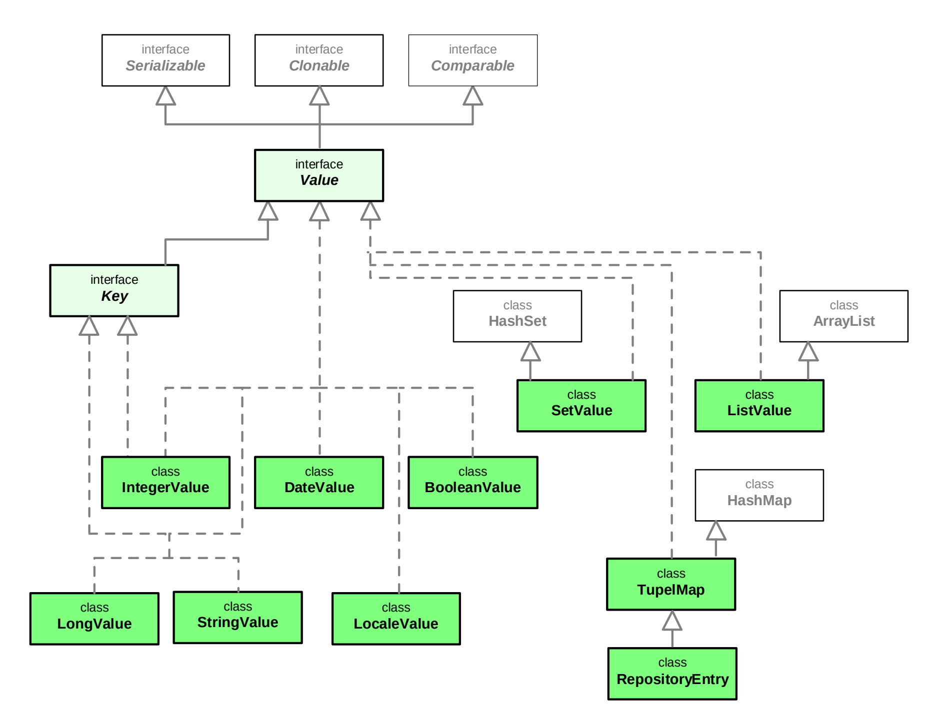


Abb. 8 – Klassendiagramm – Interfaces `Key` und `Value`

Die Objektdaten können auch als Behälter von Attribut-Wert-Paaren repräsentiert werden (siehe den Abschnitt “Repräsentation als `RepositoryEntry`” auf Seite 187). Weiterhin werden Attribute von Attributmengen und Objektkategorien (siehe den Abschnitt “Attributmengen und Objektkategorien” auf Seite 191) in Form von Attribut-Wert-Paaren von VIP ContentManager verwaltet. Für die Konstruktion von Suchfiltern (siehe Abschnitt 7.6 “VIP-Objekte suchen” auf Seite 195) sind sie ebenfalls erforderlich.

Die Interfaces `Key` und `Value` sind Bestandteil des Packages `de.gauss.lang`. Dort befinden sich konkrete Implementierungen dieser Interfaces, die Realisierungen für Attributwerte darstellen. Die folgende Liste stellt die in VIP ContentManager verwendeten, elementaren Klassen für die Attributwerte zusammen:

- `StringValue`
- `LongValue`
- `BooleanValue`
- `DateValue`
- `IntegerValue`
- `LocaleValue`
- `ListValue`
- `SetValue`

Dabei implementiert u.a. die Klasse `StringValue` das Interface `Key`.

```
public interface Value extends Comparable, Cloneable, Serializable
{
   public String toString();
   public Object clone();
   public Object getValue();
   public boolean isNull();
   public String format(Locale locale);
}
```

```
public interface Key extends Value
{
   public String getKey();
}
```

Repräsentation als `RepositoryEntry`

Die Daten eines VIP-Objekts können auch als Attribut-Wert-Paare verarbeitet werden. Die Klasse `RepositoryEntry` repräsentiert einen Behälter (Interface `java.util.Map`), der Schlüssel (Interface `Key`) auf Werte abbildet (Interface `Value`). Objekte der Klasse `RepositoryEntry` sind insbesondere für VIP PortalManager von elementarer Bedeutung.

Die Klasse `RepositoryEntry` ist Bestandteil des Packages `de.gauss.vip.repository`. Die Verwendung ist analog zu der eines Objekts des Interfaces `Map` aus dem Java 2 SDK. Detaillierte Informationen zu dieser Klasse erhalten Sie in der Online-Dokumentation.

```
public class RepositoryEntry implements Serializable, Cloneable
{
   public RepositoryEntry(int initialCapacity, float loadFactor)
   public RepositoryEntry(int initialCapacity)
   public RepositoryEntry()
   public RepositoryEntry(Map map)
   public RepositoryEntry(TupelMap map)
   public RepositoryEntry(Attributes attributes) throws NamingException
   public RepositoryEntry(Attributes attributes, String dn)
      throws NamingException
   public synchronized RepositoryEntryIterator getIterator();
   public synchronized Object getValue()
   public Object clone()
   public synchronized Attributes getAsAttributes()
   public synchronized RepositoryEntryIterator getSortedIterator(int
      direction);
}
```

Die Objektdaten eines VIP-Objekts können mit den folgenden Methoden als `RepositoryEntry`-Objekt dargestellt werden, bzw. die Objektdaten können über die Attribute dieses Objekts gesetzt werden.

```
public interface ObjectData
{
   ...
   public RepositoryEntry getAsRepositoryEntry(Collection attributes,
      String dplSystem);
   public void setFromRepositoryEntry(RepositoryEntry entry);
}
```

Die Konstanten für die verwendeten Attribute sind im Interface `FieldNames` definiert. Die folgende Tabelle zeigt die Beziehung zwischen dem Attributnamen (als Zeichenkette), der verwendeten Konstanten und der Klasse des Attributwerts. Für Behälterklassen werden zusätzlich die Klassen für die Elemente des Behälters in Klammern angegeben. Weitere Eigenschaften von Attributen sind:

- ob sie den Wert `null` erlauben
- ob man nach ihnen suchen kann, d.h., ob sie für die Konstruktion von Filtern verwendet werden können
- ob die Attributwerte verändert werden können

Tabelle 13 – Standardmetadaten

Attributname	Attribut-konstante	Klasse des Werts	Nullwerte	Suchbar	Modifizierbar
“acl”	`ACL`	`Acl`	✓	✗	✓
“all_children”	`CHILDREN`	`ListValue [ObjectId]`	✗	✗	✗

Attributname	Attribut-konstante	Klasse des Werts	Nullwerte	Suchbar	Modifizierbar
“category”	CATEGORY	StringValue	✓	✓	✓
“created_by”	CREATED_BY	User	✗	✓	✗
“date_created”	DATE_CREATED	DateValue	✗	✓	✗
“date_expire”	DATE_EXPIRE	DateValue	✓	✓	✓
“date_modified”	DATE_MODIFIED	DateValue	✓	✓	✗
“date_released”	DATE_RELEASED	DateValue	✓	✓	✗
“date_released_at”	DATE_PENDING_RELEASE	DateValue	✓	✓	✓
“deployment_hint”	DEPLOYMENT_HINT	StringValue	✗	✓	✓
“description”	DESCRIPTION	StringValue	✓	✓	✓
“direct_release”	DIRECT_RELEASE	BooleanValue	✗	✓	✓
“email_edit”	EMAIL_EDIT	SetValue [String]	✓	✗	✓
“email_qa”	EMAIL_QA	SetValue [String]	✓	✗	✓
“email_release”	EMAIL_RELEASE	SetValue [String]	✓	✗	✓
“keywords”	KEYWORDS	SetValue [String]	✓	✗	✓
“language”	LOCALE	LocaleValue	✓	✓	✓

Attributname	Attribut-konstante	Klasse des Werts	Nullwerte	Suchbar	Modifizierbar
“linked_from”	LINKED_FROM	SetValue [ObjectId]	✗	✗	✗
“links_to”	LINKS_TO	SetValue [Link]	✗	✗	✓
“modified_by”	MODIFIED_BY	User	✓	✓	✗
“oid”	OID	ObjectId	✗	✓	✗
“pathname”	PATHNAME	StringValue	✓	✗	✗
“released_by”	RELEASED_BY	User	✓	✓	✗
“state”	STATE	ObjectState	✗	✓	✗
“subtitle”	SUBTITLE	StringValue	✓	✓	✓
“surrogate_url”	SURROGATE_URL	StringValue	✓	✗	✗
“target_group”	TARGET_GROUP	StringValue	✓	✓	✓
“title”	TITLE	StringValue	✗	✓	✓
“topic”	TOPIC	ObjectId	✓	✓	✗
“type”	TYPE	ObjectType	✗	✓	✓
“url”	URL	StringValue	✓	✗	✗
“version”	VERSION	Version	✗	✓	✗
“template”	TEMPLATE	ObjectId	✓	✓	✓
“website”	WEBSITE	StringValue	✗	✗	✗

Wird die Methode `getAsRepositoryEntry` mit `null` als Argument für die Attributsammlung angegeben, werden alle Attribute im `RepositoryEntry`-Objekt gesetzt. Insbesondere werden auch alle Attribute der Attributmenge des Objekttyps und alle Eigenschaften der Objektkategorie übernommen. Das Setzen der Objektdaten berücksichtigt ebenfalls diese Attribute und Eigenschaften.

Der Parameter für das Deploymentsystem hat nur Bedeutung für die URL und den Pfadnamen, da diese Attribute vom Deploymentsystem abhängig sind. Wenn diese Attribute nicht benötigt werden, kann `null` übergeben werden.

Attributmengen und Objektkategorien

Die Attributverwaltung kann über `VipRuntime.getAttributeHandler(website)` ermittelt werden.

Über dieses Interface ist es möglich, alle für eine Website verwendeten Attributmengen und Objektkategorien auszulesen.

Attributmengen

In VIP ContentManager wird unter einer Attributmenge eine durch einen Namen identifizierbare Menge verstanden, deren Elemente so genannte Attributdefinitionen sind. Eine Attributdefinition enthält dabei alle wesentlichen Eigenschaften eines Attributs, also z.B. den eindeutigen Bezeichner, die verwendete Java-Klasse für den Attributwert oder die maximale Anzahl von Zeichen für Textattribute.

Jedes VIP-Objekt verfügt über eine Menge von Standardattributen, die Standardmetadaten (siehe Tabelle 13 “Standardmetadaten” auf Seite 188). Die Standardattribute eines VIP-Objekts können mit der Methode `getStandardAttributes` des Interfaces `AttributeHandler` bestimmt werden. Die Definition von Standardattributen wird mit der Methode `getStandardAttributeDefinition(Key key)` ausgelesen.

Zusätzlich zu den Standardmetadaten können VIP-Objekte mit Attributen aus Attributmengen ausgestattet werden. Voraussetzung ist, dass der zugrunde liegende Objekttyp mit einer Attributmenge verknüpft wurde. Dies kann über das VIP-Administrationsprogramm erfolgen (siehe VIP ContentManager-Administratorhandbuch). Wichtig ist, dass Attributmengen nur im Zusammenhang mit dem Objekttyp verwendet werden können. Zum Beispiel kann jedes VIP-Objekt des Typs `HTML` Eigenschaften haben, die nur für HTML-Dokumente eine Bedeutung besitzen. Die Attributmenge eines Objekttyps kann mit der Methode `getAttributeSet` des Interfaces `ObjectType` bestimmt werden.

Objektkategorien

Auch eine Objektkategorie ist eine benannte Menge von Attributdefinitionen. Es wird jedoch nicht der Begriff Attributmenge verwendet, da Objektkategorien in einem anderen Kontext eingesetzt werden. Jedem VIP-Objekt in einer Website kann eine Objektkategorie zugeordnet werden. Diese Zuordnung hat – im Unterschied zur Attributmenge – nichts mit dem Objekttyp des VIP-Objekts zu tun. Zum Beispiel kann es eine Objektkategorie “Rechnung” geben, die typische Eigenschaften einer Rechnung als Attribute definiert (z.B. “Rechnungsstatus”). Diese Attribute können für beliebige Objekttypen einen Wert haben (z.B. für HTML-Dateien ebenso wie für Microsoft-Word-Dateien). Einem VIP-Objekt wird mit der Methode `setObjectCategory` aus dem Interface `ObjectData` eine Objektkategorie zugeordnet.

```
public interface AttributeHandler
{
    public Website getWebsite();
    public List getAttributSets();
    public AttributSet getAttributSet(String attrSetName);
    public List getStandardAttributes() throws VipApiException;
    public AttributeDefinition getStandardAttributeDefinition(Key key)
           throws VipApiException;
    public List getObjectCategories();
    public ObjectCategory getObjectCategory(String objCategoryName);
}
```

Eine Attributmenge und eine Objektkategorie unterscheiden sich formal nicht voneinander. Sie haben einen eindeutigen Namen und enthalten eine Liste von Attributdefinitionen.

```
public interface AttributeSet
{
   public String getName();
   public List getAttributes();
   public AttributeDefinition getDefinition(Key key);
}

public interface ObjectCategory
{
   public String getName();
   public List getAttributes();
   public AttributeDefinition getDefinition(Key key);
}
```

Die Definition eines Attributs wird durch ein Objekt des Interfaces `AtttributeDefinition` repräsentiert. Folgende Eigenschaften charakterisieren ein in VIP ContentManager verwendetes Attribut.

- der Klassenname für die Java-Klasse, die den Attributwert repräsentiert (`Value`-Interface)

  Es sind folgende Klassen für Attribute möglich: `StringValue`, `DateValue`, `IntegerValue`, `LongValue`, `ListValue`, `BooleanValue` und `SetValue`.

- ein eindeutiger Bezeichner für das Attribut (als Objekt des Interfaces `Key`)
- eine lokalisierte Beschreibung des Attributs
- die Feldlängenbegrenzung (nur für `StringValue`-Klassen relevant)

```
public interface AttributeDefinition
{
   public Key getKey();
   public int getMaxStringLength();
   public String getMessage(Locale locale);
   public String getValueClassName();
}
```

7.6 VIP-Objekte suchen

Die Suche nach VIP-Objekten wird durch die Methode `ObjectHandler.filter` ermöglicht. Eine Suchbedingung wird dabei als Objekt vom Typ `Filter` repräsentiert. Mit diesen Filterobjekten lassen sich logische Ausrücke modellieren, die sich zu dem Wahrheitswert `true` oder `false` auswerten lassen. Dabei können mithilfe von logischen Verknüpfungen auch komplexe Filter zusammengestellt werden.

Grundsätzlich gibt es drei verschiedene Kategorien von Filterklassen.

- Filter, die auf den Attributen der VIP-Objekte basieren (zu den Attributen siehe Tabelle 13 “Standardmetadaten” auf Seite 188). Diese Filter erweitern die abstrakte Klasse `AttributeFilter`.
- Filter, die schon vorhandene Filter logisch miteinander verknüpfen und dadurch die Konstruktion von komplexen Suchanfragen ermöglichen. Dabei wird zwischen Filtern mit einem bzw. mit zwei Argumenten unterschieden (Oberklasse `UnaryFilter` bzw. `BinaryFilter`). Folgende Filterklassen gehören zu dieser Kategorie: `NotFilter`, `AndFilter`, `OrFilter`.
- Filter, die eine vordefinierte Suchfunktion ausführen: `RootTemplateFilter`, `SubtreeFilter` und `PrincipalFilter`.

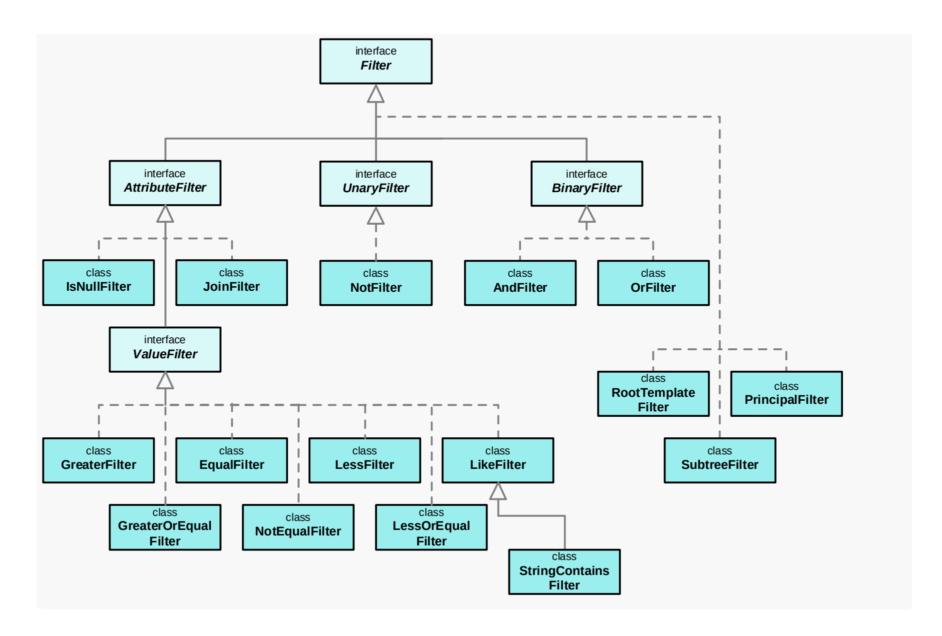


Abb. 9 – Klassendiagramm – Filterklassen und -Interfaces

Attributfilter

VIP ContentManager verwaltet die Daten einer Website (einschließlich der Inhalte und der Metadaten) in einer Datenbank. Bei der Suche wird – technisch gesehen – auf Basis des Filterobjekts eine für das jeweilige Datenbanksystem geeignete Suchanfrage (eine SQL-Anweisung der Form `SELECT ... FROM ... WHERE ... ORDER BY ...`) formuliert. Es gibt jedoch einige Einschränkungen bezüglich der Verwendung von Attributen für die Formulierung von SQL-Anfragen. Aus diesem Grund können nicht alle Attribute in einer Suche verwendet werden. Genaue Angaben zu den suchbaren Attributen finden Sie in Tabelle 13 “Standardmetadaten” auf Seite 188.

Anhand der folgenden Tabelle kann abgelesen werden, welche Attributtypen mit welchen Filtern zusammen verwendet werden können. Die Konstanten der Attribute sind in den Interfaces `FieldNames` und `SearchableKeys` definiert. Beide Interfaces sind Bestandteil des Packages `de.gauss.vip.api.object`.

Tabelle 14 – Attributtypen und Filter

	`LikeFilter`	`StringContainsFilter`	`IsNullFilter`[a]	`JoinFilter`[b]	`EqualFilter`	`NotEqualFilter`	`GreaterFilter`	`GreaterOrEqualFilter`	`LessFilter`	`LessOrEqualFilter`
`IntegerValue`, `LongValue`			✓	✓	✓	✓	✓	✓	✓	✓
`ListValue`[c], `SetValue`[c]	✓	✓	✓							
`DateValue`			✓	✓	✓	✓	✓	✓	✓	✓
`BooleanValue`			✓	✓	✓	✓				
`StringValue`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`LocaleValue`[d], `ObjectId`, `ObjectType`, `ObjectState`, `User`[d], `Version`					✓					

(a) Mit diesem Filter kann getestet werden, ob der Wert für ein Attribut gesetzt wurde.

(b) Dieser Filter vergleicht zwei Attributwerte miteinander. Beide Werte müssen vom gleichen Typ sein.

(c) Gesucht wird jeweils nach einem Eintrag aus der Liste, der als `StringValue` angegeben wird.

(d) Bei `LocaleValue` und `User` kann der Wert auch als `StringValue` angegeben werden, dadurch können auch die anderen Filter verwendet werden.

Beispiel

```
new IsNullFilter(FieldNames.SUBTITLE)

new StringContainsFilter(FieldNames.EMAIL_QA,
      new StringValue("jstein@company.com"))

new EqualFilter(FieldNames.CREATED_BY,
   VipRuntime.getAdminHandler().getUser("jstein"))

new EqualFilter(FieldNames.CREATED_BY, new StringValue("jstein"))
```

Vordefinierte Suchfunktionen als Filter

- Die Klasse `RootTemplateFilter` sucht nach allen Vorlagenobjekten einer Website, die selbst keine Vorlage besitzen.
- Mit der Klasse `SubtreeFilter` können alle VIP-Objekte unterhalb eines gegebenen VIP-Objekts (Thema in der Themenstruktur bzw. Vorlage in der Vorlagenstruktur) ermittelt werden. Der `SubtreeFilter` kann auch ohne Angabe des `Topology`-Parameters verwendet werden. In diesem Fall gilt die Themenstruktur als Defaultwert.

- Die Klasse `PrincipalFilter` sucht nach VIP-Objekten, denen ein bestimmter Principal (Benutzer, Gruppe oder Rolle) zugeordnet ist. Zusätzlich besteht die Möglichkeit, das Ergebnis auf VIP-Objekte einzuschränken, für die der Principal ein bestimmtes Recht (`VipObjectPermission`) besitzt. Weitere Informationen zu Principals und Zugriffsrechten finden Sie in Abschnitt 7.7 “Zugriffssteuerung” ab Seite 204.

Beispiel

Mit dem folgenden Filter werden alle VIP-Objekte (einschließlich des spezifizierten Startknotens) zurückgegeben, die in der Themenstruktur unter dem VIP-Objekt mit der OID “4711” liegen.

```
new SubtreeFilter(new de.gauss.vip.api.lang.ObjectId("4711"))
```

Zusätzliche Suchattribute

Das Interface `SearchableKeys` erweitert das Interface `FieldNames` um Konstanten, die nicht direkt einem Attribut zugeordnet sind.

Mit diesen Attributen werden zusätzliche Suchmöglichkeiten geschaffen, oder bestehende Suchausdrücke können einfacher aufgebaut werden.

ACL_OWNER

Mit diesem Attribut kann festgestellt werden, ob ein VIP-Objekt eine eigene Zugriffssteuerungsliste (ACL) besitzt oder diese nur erbt.

Klasse des Attributwerts

`ObjectId`

Beispiel

Der folgende Filter sucht alle VIP-Objekte mit einer eigenen Zugriffssteuerungsliste.

```
new JoinFilter(FieldNames.OID, SearchableKeys.ACL_OWNER)
```

STATE_NAME

Mit diesem Attribut kann nach VIP-Objekten gesucht werden, die einen bestimmten Objektstatus haben.

Klasse des Attributwerts

`StringValue`

Beispiel

```
new EqualFilter(SearchableKeys.STATE_NAME,
  new StringValue(ObjectState.RELEASED))
```

ist äquivalent zu

```
new EqualFilter(FieldNames.STATE,
  VipRuntime.getObjectHandler(website)
    .getWebsite().getObjectState(ObjectState.RELEASED))
```

TYPE_NAME

Mit diesem Attribut kann nach bestimmten Objekttypen gesucht werden.

Klasse des Attributwerts

`StringValue`

Beispiel

```
new EqualFilter(SearchableKeys.TYPE_NAME,
  new StringValue(ObjectType.HTML))
```

ist äquivalent zu

```
new EqualFilter(FieldNames.TYPE,
  VipRuntime.getObjectHandler(website).getWebsite().getObjectType
    (ObjectType.HTML))
```

Suchparameter

In den `filter`-Methoden werden neben dem Filterobjekt verschiedene andere Parameter für die Suche spezifiziert. Im Folgenden wird eine der vier `filter`-Methoden beispielhaft beschrieben.

```
public List filter(ContextId cid, Filter filter, ObjectId startOID,
   List sortList, int startResult, int numberOfResults)
   throws VipApiException;
```

- `startOID` – Durch die Angabe eines Startknotens erfolgt die Suche erst unterhalb des angegebenen VIP-Objekts. In diesem Fall werden der Startknoten selbst (`startOID`) sowie alle in der Themenstruktur untergeordneten VIP-Objekte berücksichtigt. Das Suchergebnis ist eine Liste von OIDs. Die `startOID` muss ein Themenobjekt referenzieren.

  Tipp: Um die gesamte Website zu durchsuchen, übergeben Sie `null` als `startOID`. In diesem Fall wird der Navigationsbaum nicht berücksichtigt, da ohnehin sämtliche VIP-Objekte in die Suche einbezogen sind. Wenn Sie hingegen die OID des Wurzelknotens als `startOID` übergeben, muss der gesamte Navigationsbaum durchlaufen werden. Das dauert erheblich länger.

- `sortList` – Hier kann eine Liste von Attributkonstanten angegeben werden, nach denen die Ergebnismenge sortiert werden soll. Standardmäßig werden die gefundenen VIP-Objekte nach der OID sortiert.
- `numberOfResults` – Durch diesen Parameter kann die maximale Anzahl der VIP-Objekte, die in der Ergebnisliste enthalten sein soll, bestimmt werden. Mit dem Wert `-1` wird das gesamte Suchergebnis zurückgegeben.

Beispiel

Das folgende Beispiel sucht nach allen VIP-Objekten, die nach einem vorgegebenen Datum angelegt wurden und sich im Objektstatus “freigegeben” oder “verzögert freigegeben” befinden (`RELEASED` oder `PENDING_RELEASE`).

```
public List filterDemo(ContextId cid,String websiteName,Date createDate)
{
   ObjectHandler oh = VipRuntime.getObjectHandler(websiteName);
   Filter f1 = new GreaterFilter(FieldNames.DATE_CREATED,
      new DateValue(createDate));
   Filter f2 = new OrFilter(
      new EqualFilter(SearchableKeys.STATE_NAME,
         new StringValue(ObjectState.RELEASED)),
      new EqualFilter(SearchableKeys.STATE_NAME,
         new StringValue(ObjectState.RELEASE_PENDING));

   Filter f = new AndFilter(f1,f2);
   List l = oh.filter(cid, f, null, null, 0, -1);
   return l;
}
```

7.7 Zugriffssteuerung

Principals

Der Administrationsserver verwaltet u.a. Benutzer, Gruppen und Rollen. Diese “Rechteträger” werden im VIP Java API durch Objekte repräsentiert, die das Interface `Principal` implementieren. Jeder Principal hat einen eindeutigen Namen und einen Typ. Die Methoden `getUser`, `getGroup` und `getRole` des `AdminHandler`-Interfaces liefern ein entsprechendes `Principal`-Objekt zurück.

```
public interface Principal extends Value
{
   public static final int TYPE_USER;
   public static final int TYPE_GROUP;
   public static final int TYPE_ROLE;
   public static final int TYPE_GROUPROLE;
   public static final int TYPE_WORLD;
   public String getName();
   public int getType();
}
```

Der Typ `TYPE_GROUPROLE` repräsentiert eine Gruppenrolle, die durch ein Objekt der Klasse `GroupRole` angelegt werden kann. Gruppenrollen werden nicht im VIP-Administrationsprogramm verwaltet, sondern sind für die Verwendung in Zugriffssteuerungslisten für VIP-Objekte gedacht. Eine Gruppenrolle definiert die Schnittmenge aus den Benutzern einer Gruppe und einer Rolle: Zu der Gruppenrolle gehören diejenigen Benutzer, die sowohl der Gruppe als auch der Rolle zugeordnet sind (weitere Informationen dazu finden Sie im VIP ContentManager-Benutzerhandbuch).

```
public class GroupRole implements Principal
{
   public GroupRole(String groupName, String roleName);
   public String getGroupName();
   public String getRoleName();
}
```

Der Typ `TYPE_WORLD` repräsentiert den vordefinierten Benutzer “Jeder” und wird ebenfalls nur für die Definition von Zugriffsrechten in den Zugriffssteuerungslisten der VIP-Objekte benötigt. Es gibt in einem VIP-CM-System nur ein einziges Objekt der zugehörigen Klasse `World` (Singleton-Muster).

```
public class World implements Principal
{
   static public World getInstance();
}
```

Für die Benutzerverwaltung sind insbesondere die Typen `TYPE_USER`, `TYPE_GROUP` und `TYPE_ROLE` relevant. Für diese gibt es drei zugehörige Klassen: `User`, `Group` und `Role`. Identifiziert werden Benutzer durch eine Benutzerkennung, Gruppen und Rollen durch einen eindeutigen Namen. Die weiteren Eigenschaften (“Profile”) werden über das Interface `AdminHandler` ermittelt. Jede der drei Klassen bietet eine Methode `hasProfile` – diese prüft, ob die weiteren Eigenschaften überhaupt vorhanden sind.

> **Hinweis:** Über das Interface `AdminHandler` haben Sie auch die Möglichkeit, neue Principals anzulegen, Profile zu bearbeiten und Zuordnungen von Principals vorzunehmen. Ausführliche Informationen zum `AdminHandler`-Interface enthält Kapitel 4 “Administrationsinterface”.

Zugriffssteuerungslisten – das Interface ACL

Jedes VIP-Objekt verfügt über eine Zugriffssteuerungsliste (ACL). In der Zugriffssteuerungsliste ist festlegt, welche Benutzer, Gruppen und Rollen mit welchen Rechten auf das VIP-Objekt zugreifen dürfen. Diese Zugriffssteuerungslisten werden durch das Interface `Acl` (`de.gauss.vip.api.lang`) modelliert.

Eine Zugriffssteuerungsliste ordnet einem `Principal`-Objekt (Benutzer, Gruppe, Rolle, Gruppenrolle oder Jeder) eine Menge von positiven (erlaubten) bzw. negativen (nicht erlaubten) Rechten zu. Die Rechte werden durch Objekte des Interfaces `VipObjectPermission` modelliert (siehe Tabelle 4 “Konstanten der Zugriffsrechte für VIP-Objekte” auf Seite 86).

Die Zugriffssteuerungsliste eines VIP-Objekts wird mit der Methode `getACL` aus dem Interface `ObjectData` ermittelt. Die Methode `setACL` ändert die Zugriffssteuerungsliste, wobei die Änderung erst in Kraft tritt, wenn das VIP-Objekt über die Methode `ObjectHandler.change` geändert wird. Um die Zugriffssteuerungsliste eines VIP-Objekts zu ändern, ist ein spezielles Recht erforderlich (`VipObjectPermission.CHANGE_RIGHTS`).

Zugriffssteuerungslisten können vererbt sein. Wenn es für ein bestimmtes VIP-Objekt keine eigenen (objektspezifischen) Zugriffseinstellungen gibt, erbt das VIP-Objekt die Zugriffssteuerungsliste des übergeordneten Themas. Das Thema selbst kann wiederum eine geerbte Zugriffssteuerungsliste haben usw. Die Methode `isInherited` prüft, ob es sich um eine vererbte Zugriffssteuerungsliste handelt. In diesem Fall liefert `getOwner` das VIP-Objekt, das die Zugriffssteuerungsliste besitzt.

Die Methode `checkPermission` prüft, ob ein vorgegebener Benutzer ein bestimmtes Recht hat (bezogen auf die Zugriffssteuerungsliste eines VIP-Objekts). Wenn der Zugriff auf das Benutzerprofil möglich ist, dann werden auch die Gruppen und Rollen des Benutzers überprüft (die Methode `hasPermission` prüft nur die direkt dem Principal zugeordneten Rechte). Ein über die Kontextverwaltung angemeldeter Benutzer hat immer ein Profil (siehe auch Abschnitt 6.1 “An- und Abmelden” auf Seite 134). Die weiteren Methoden dienen dazu, Rechte für ein gegebenes `Principal`-Objekt hinzuzufügen bzw. zu entfernen.

```
public interface Acl extends Value
{
    public static final boolean GRANT ;
    public static final boolean DENY ;
    public boolean isInherited();
    public ObjectId getOwner();
    public boolean checkPermission (User user,
        VipObjectPermission permission);
    public boolean hasPermission (Principal principal,
        VipObjectPermission permission);
    public void addPermission (Principal principal,
        VipObjectPermission permission, boolean policy);
    public void removePermission (Principal principal,
        VipObjectPermission permission, boolean policy);
    public void removePrincipalPermissions (Principal principal);
    public Set getPrincipals ();
    public Set getPermissions (Principal principal,boolean policy);
}
```

KAPITEL 8

Deployment

Die Deploymentsysteme erzeugen aus den in der Datenbank gespeicherten VIP-Objekte Dateien, die über einen Browser angezeigt werden können. Diese Dateien werden als Web-Objekte bezeichnet. Allgemeine Informationen zum Deployment enthält der Abschnitt 2.3 “Seitengenerierung (Deployment)” ab Seite 32.

Das Interface `DeploymentHandler` bietet Zugriff auf die Deploymentsysteme einer Website. Es stellt folgende Funktionen zur Verfügung:

- Abrufen von Deployment-Metadaten (u.a. URL und Pfad) zu einem Web-Objekt innerhalb eines Deploymentsystems

  Siehe Abschnitt 8.1 “Deployment-Metadaten” ab Seite 211

- Abfrage von Statusinformationen zu einem Deploymentsystem und zu laufenden Transaktionen innerhalb eines Deploymentsystems

  Siehe Abschnitt 8.2 “Informationen zum Status des Deployments” ab Seite 213

- Abrufen von Fehlerlisten und -meldungen, die bei einer laufenden Transaktion innerhalb eines Deploymentsystems aufgetreten sind, sowie von Fehlermeldungen, die zu einem bestimmten VIP-Objekt innerhalb eines Deploymentsystems aufgetreten sind

  Siehe Abschnitt 8.3 “Deploymentfehler” ab Seite 218

Eine Instanz des `DeploymentHandler`-Interfaces erhalten Sie über `VipRuntime.getDeploymentHandler`.

Eine kurze Beschreibung der einzelnen Methoden des `DeploymentHandler`-Interface wird in den folgenden Abschnitten gegeben. Soweit nicht anders angegeben, befinden sich alle Klassen und Interfaces im Package `de.gauss.vip.api.deployment`.

```
package de.gauss.vip.api.deployment
public interface DeploymentHandler
{

//information on the meta data of web objects belonging to a certain
//deployment system
public DeploymentMetaData get(ContextId cid, String dplName,
                              ObjectId oid);
public ObjectId getOIDByURL(ContextId cid, String dplName, String url);

// status information on deployment systems
public DeploymentStatus getStatus(ContextId cid, String dplName,
                                  Long transactionId);
public boolean isVisible(String dplName);
public boolean isRunning(String dplName);

// information on deployment errors
public List getErrors(ContextId cid, String dplName, ObjectId oid);
public List getErrors(ContextId cid, String dplName,
                      Long transactionId);

public String getWebsiteName();
public DeploymentSystem getDeploymentSystem(String dplName);
public Set getDeploymentSystems();
}
```

8.1 Deployment-Metadaten

Zu den Deployment-Metadaten gehören u.a. die URL und der Pfad zum generierten Web-Objekt. Für jedes VIP-Objekt werden – abhängig von der Anzahl installierter Deploymentsysteme – mehrere Web-Objekte erzeugt. Die Deployment-Metadaten dieser Web-Objekte unterscheiden sich voneinander, da die Web-Objekte für unterschiedliche Workflow-Sichten (Edit, QS oder Produktion) erzeugt werden können. Außerdem können sich die VIP-CM-Server, zu denen die Deploymentsysteme gehören, auf verschiedenen Rechnern befinden.

Die Deployment-Metadaten eines Web-Objekts können über den `DeploymentHandler` mit der Methode `get` angefordert werden. Dabei werden neben Kontext-ID eines angemeldeten Benutzers die OID des zugehörigen VIP-Objekts und der Name des Deploymentsystems (als String) übergeben. Die zurückgelieferten Deployment-Metadaten werden durch das Interface `DeploymentMetaData` repräsentiert:

```
public interface DeploymentMetaData
{
//Information about the URLs of a web object
public String getURL();
public String getSurrogateURL();
public String getStaticURL();

//Information about the path to the web object
public String getPath();
public String getSurrogatePath();
public String getStaticPath();


public String getDeploymentSystem();
public ObjectId getOID();
}
```

Mithilfe der Methoden des `DeploymentMetaData`-Interfaces können die URL und der Pfad zum generierten Web-Objekt sowie (falls vorhanden) zur Surrogatseite bzw. statifizierten Seite ermittelt werden. Ob eine Surrogatseite erzeugt wird, hängt vom Objekttyp ab. Die Statifizierung wird in den Einstellungen des Deploymentsystems konfiguriert; dies erfolgt über das VIP-Administrationsprogramm.

Falls das Objekt kein Surrogat besitzt, geben sowohl `getSurrogateURL` als auch `getSurrogatePath` `null` zurück. Analog dazu geben die Methoden `getStaticURL` und `getStaticPath` `null` zurück, wenn das Objekt nicht statifiziert wurde.

Das folgende Beispiel zeigt, wie Deployment-Metadaten zum Wurzelobjekt einer Website abgerufen werden.

```
//login
ContextId cid = VipRuntime.getContextHandler().login( ... );

//get DeploymentHandler
DeploymentHandler deplH
               = VipRuntime.getDeploymentHandler("InternetSite");

//get and output deployment meta data for the root object (OID=1)
ObjectId oidRoot = new ObjectId("1");
DeploymentMetaData rootDplMD = deplH.get(cid, "InternetSite_edit",
                                         oidRoot);
System.out.println(" root obj URL  : " + rootDplMD.getURL() );
System.out.println(" root obj Path : " + rootDplMD.getPath() );
```

8.2 Informationen zum Status des Deployments

Über den `DeploymentHandler` lassen sich Statusinformationen zu Deploymentsystemen und darin laufenden Transaktionen abfragen. Die Statusinformationen sind im Interface `DeploymentStatus` gekapselt.

> **Hinweis:** Die Informationen, die über das Interface `DeploymentStatus` abgerufen werden können, stehen im VIP-Administrationsprogramm über *Systemverwaltung* ⟶ *Laufende Server* ⟶ *{Servername}* ⟶ *Berichte* ⟶ *DeploymentSystemHandler* zur Verfügung. Informationen zu den Angaben des Berichts enthält das VIP ContentManager-Administratorhandbuch.

Das Interface `DeploymentStatus` kann über die Methode `getStatus` des `DeploymentHandler`-Interfaces abgerufen werden kann. Neben der Kontext-ID eines angemeldeten Benutzers und dem Namen des Deploymentsystems ist beim Aufruf der Methode die Transaktions-ID einer Transaktion anzugeben, für die Statusinformationen abgerufen werden sollen. Wird stattdessen `null` übergeben, enthält das zurückgelieferte `DeploymentStatus`-Objekt Informationen zu allen laufenden Transaktionen des Deploymentsystems.

Das Interface `DeploymentStatus` ist folgendermaßen aufgebaut:

```
public interface DeploymentStatus
{
//information on analyzed and processed objects
public List getAnalysingOidList();
public String getAnalysingMessage(Locale locale);
public List getExecutingOidList();
public String getExecutingMessage(Locale locale);

//information on deployment orders and deployment jobs
public long getNoOfAnalysedOrders();
public long getNoOfWaitingOrders();
public long getNoOfExecutedJobs();
public long getNoOfWaitingJobs();
```

```
//information on waiting and execution time of orders and jobs
public double getAvgOrderWaitingTime();
public double getAvgOrderServingTime();
public double getAvgJobWaitingTime();
public double getAvgJobServingTime();

//information on transactions
public List getRunningTransactions();
public Long getTransactionId();
public Date getStartDate();
}
```

Hinweis: Falls zu einer Transaktion ein `Progress`-Objekt angefordert wurde, kann man über dieses Objekt auch die Transaktions-ID erfragen, siehe “Transaktionskontrolle – das Interface `Progress`” auf Seite 167. Alternativ dazu können die IDs aller in einem Deploymentsystem laufenden Transaktionen über die Methode `getRunningTransactions` des `DeploymentStatus`-Interfaces abgerufen werden.

Über die Methoden des Interfaces `DeploymentStatus` können folgende Informationen zu einem Deploymentsystem abgerufen werden:

- Statusinformationen zur Auftragsanalyse (order analysis)
- Statusinformationen zur Ausführung der Deployment-Jobs (job execution)

Auftragsanalyse

Nach jeder Änderung von VIP-Objekten müssen die Deploymentsysteme neue Web-Objekte erzeugen, die den aktuellen Stand der VIP-Objekte widerspiegeln. Bei jeder Änderung eines VIP-Objekts wird deshalb ein Deploymentauftrag (deployment order) erstellt. Die Analyse dieses Auftrags umfasst folgende Aufgaben:

- Es wird überprüft, ob die Deployment-Metadaten des entsprechenden Web-Objekts (URL, Pfad etc.) angepasst werden müssen.
- Es wird geprüft, ob andere Web-Objekte von der Änderung betroffen sind (z.B. wenn eine Vorlage bearbeitet wurde).
- Für alle neu zu generierenden Web-Objekte werden so genannte Deployment-Jobs erstellt, d.h. Anweisungen zur Erzeugung eines Web-Objekts innerhalb eines Deploymentsystems.

Ausführung der Deployment-Jobs

In dieser Phase werden die erstellten Jobs abgearbeitet und die von der Änderung des VIP-Objekts betroffenen Web-Objekte neu erzeugt. Die erzeugten Dateien (HTML-Seiten, Bilder usw.) werden im Dateisystem des VIP-CM-Servers gespeichert, auf dem das Deploymentsystem installiert ist.

Verfügbare Statusinformationen

Für beide Phasen des Deployments können über das Interface `DeploymentStatus` folgende Informationen abgerufen werden:

- Anzahl der auf Bearbeitung wartenden Aufträge bzw. Jobs: `public long getNoOfWaitingOrders` und `public long getNumberOfWaitingJobs`
- Anzahl der analysierten Aufträge bzw. abgearbeiteten Jobs: `public long getNoOfAnalysedOrders` und `public long getNoOfExecutedJobs`
- durchschnittliche Bearbeitungszeit für einen Auftrag bzw. Job: `public double getAvgOrderServingTime` und `public double getAvgJobServingTime`

- durchschnittliche Wartezeit für einen Auftrag bzw. Job: `public double getAvgOrderWaitingTime` und `public double getAvgJobWaitingTime`

Wurde der Deployment-Status für eine bestimmte Transaktion angefordert, d.h. beim Aufruf von `DeploymentHandler.getStatus` eine Transaktions-ID angegeben, können zusätzlich Listen mit den OIDs der VIP-Objekte abgerufen werden, deren Web-Objekte innerhalb dieser Transaktion gerade analysiert bzw. bearbeitet werden. Zu diesem Zweck stehen die Methoden `getAnalysingOidList` und `getExecutingOidList` zur Verfügung. Außerdem können für die Transaktion Meldungen (als String) abgerufen werden, die den Fortschritt der Analyse bzw. Job-Ausführung beschreiben. Dazu dienen die Methoden `getAnalysingMessage` und `getExecutingMessage`.

Wurde der Deployment-Status ohne Angabe einer Transaktions-ID angefordert, liefern diese Methoden `null` zurück. Nach Beenden einer Transaktion stehen keine Statusinformationen mehr zur Verfügung.

Beispiel

Das folgende Beispiel zeigt, wie die OIDs der VIP-Objekte ausgegeben werden, deren Web-Objekte gerade analysiert bzw. bearbeitet werden. Auch die dazu gehörigen Meldungen werden abgerufen. Die Statusinformationen betreffen alle in einem Deploymentsystem laufenden Aktionen, d.h, es wird keine Transaktions-ID übergeben.

```
//login and get DeploymentHandler
ContextId cid = VipRuntime.getContextHandler().login( ... );
DeploymentHandler dH = VipRuntime.getDeploymentHandler("InternetSite");

//get DeploymentStatus interface for the complete deployment system,
//i.e. not for a specific transaction
DeploymentStatus ds = dH.getStatus(cid, "InternetSite_edit", null);
```

```
//get list of running transactions. Iterate through this list
//to get the DeploymentStatus objects for the individual transactions
List runningTransactions = ds.getRunningTransactions();
Iterator transactionIter = runningTransactions.iterator();
while ( transactionIter.hasNext() )
{
   Long transactionID = (Long) transactionIter.next();

   //get DeploymentStatus for the individual transactions and
   //output information
   DeploymentStatus dsTransaction = dH.getStatus(cid,
                                   "InternetSite_edit", transactionID);
   System.out.println( "oids being analysed in Transaction "+
                       dsTransaction+ " : " +
                       dsTransaction.getAnalysingOidList() );
   System.out.println( dsTransaction.getAnalysingMessage(Locale.
                                                         getDefault() ) );
   System.out.println( "oids being executed in Transaction "+
                       dsTransaction+ " : " +
                       dsTransaction.getExecutingOidList() );
   System.out.println( dsTransaction.getExecutingMessage(Locale.
                                                         getDefault() ) );
}
```

8.3 Deploymentfehler

Über das `DeploymentHandler`-Interface können auch Informationen über Fehler abgerufen werden, die während des Deployments aufgetreten sind. Zu diesem Zweck stehen zwei Methoden zur Verfügung:

- `public List getErrors(ContextId cid, String dplName, ObjectId oid)` – Diese Methode liefert Informationen über Fehler, die im angegebenen Deploymentsystem bei der Bearbeitung eines bestimmten Objekts aufgetreten sind. Das Objekt wird über seine OID spezifiziert.
- `public List getErrors(ContextId cid, String dplName, Long transactionId)` – Diese Methode liefert Informationen über Fehler, die im angegebenen Deploymentsystem während einer bestimmten Transaktion aufgetreten sind. Die Transaktion wird über eine Transaktions-ID spezifiziert.

Beide Methoden liefern als Ergebnis ein Liste zurück, die die aufgetretenen Fehler als Objekte vom Typ `DeploymentError` enthält.

Das Interface `DeploymentError` ist folgendermaßen definiert:

```
package de.gauss.vip.api.deployment;
public interface DeploymentError

{
public String getDeploymentSystem();
public ObjectId getOID();
public Long getTransactionId();
public Date getCreationDate();
public String getErrorMessage(Locale locale);
}
```

Mithilfe der Methoden des Interfaces `DeploymentError` kann die OID eines Objekts bzw. die ID einer Transaktion ermittelt werden, bei der der entsprechende Fehler aufgetreten ist (Methoden `getOID` und `getTransactionId`). Außerdem kann über die Methode `getErrorMessage` die eigentliche Fehlermeldung abgefragt werden. Die Methode `getCreationDate` ermittelt den Zeitpunkt, zu dem der Fehler aufgetreten ist.

Beispiel

Das folgende Beispiel zeigt, wie die Deploymentfehler zur aktuell laufenden Transaktion ermittelt werden.

```
//login and get DeploymentHandler
ContextId cid = VipRuntime.getContextHandler().login( ... );
DeploymentHandler dH = VipRuntime.getDeploymentHandler("InternetSite");

//get DeploymentStatus for the deployment system in order
//to retrieve the running transaction
DeploymentStatus ds = dH.getStatus(cid, "InternetSite_edit", null);
List runningTransactions = ds.getRunningTransactions();

//iterate through the running transactions in order to
//output deployment errors
Iterator transactionIter = runningTransactions.iterator();
while ( transactionIter.hasNext() )
{
   Long transactionID = (Long) transactionIter.next();

   //get list with deployment errors of the current transaction
   //iterate through the list of DeploymentError objects
   //output messages
   List dplErrors = dH.getErrors(cid, "InternetSite_edit",
                              transactionID);
   Iterator dplErrIter= dplErrors.iterator();
   while (dplErrIter.hasNext())
```

```
    {
        DeploymentError dplErr=(DeploymentError) dplErrIter.next();
        //output OID and error message
        System.out.println(" deployment error occured for oid " +
                    dplErr.getOID()+ " in transaction " + transactionID );
        System.out.println(" error message : "+
                     dplErr.getErrorMessage(Locale.getDefault()) );
    }
}
```

KAPITEL 9

Poolverwaltung

Ab Version 8.1 der VIP CM Suite können eigene Pooltypen erstellt und verwaltet werden. Auf Grundlage von eigenen Pooltypen können dann Pools mit benutzerdefinierten Verbindungen, z.B. für die Anbindung des VIP-CM-Systems an Fremdsysteme, erstellt werden. Diese Pools werden wie die Pools auf Grundlage der Standard-Pooltypen über den Pool-Mechanismus von VIP ContentManager verwaltet.

Ausführliche Informationen zu Pools und Pooltypen bietet das VIP ContentManager-Administratorhandbuch.

Zur Unterstützung der Erstellung und Verwaltung von Verbindungen, die auf eigenen Pooltypen basieren, bietet das VIP Java API folgende Elemente:

1. das Interface `de.gauss.vip.api.pool.PoolManager`, das den internen `PoolManager` von VIP ContentManager kapselt. Der `PoolManager` verwaltet die Verbindungen in Pools, d.h., er baut bei Bedarf Verbindungen auf und ab. Wird in einem Agenten eine benutzerdefinierte Verbindung benötigt, dann wird sie vom `PoolManager` angefordert.
2. die Klasse `de.gauss.vip.api.pool.PoolConnection`. Von dieser Klassen müssen Verbindungen auf Grundlage eigener Pooltypen abgeleitet werden, damit sie durch den Pool-Mechanismus von VIP ContentManager verwaltet werden können. Diese Verbindungen werden im Folgenden als benutzerdefinierte Verbindungen bezeichnet.

Wenn Sie einen Agenten entwickeln, der eine benutzerdefinierte Verbindung zu einem Fremdsystem aufbauen soll, gehen Sie folgendermaßen vor:

1. Erstellen Sie eine Klasse für die Verbindung – abgeleitet von der Basisklasse `PoolConnection`.

   Programmieren Sie die Methoden `close`, `isValid`, `execute` und `customOpen` entsprechend den Anforderungen des Fremdsystems. Wird z. B. für das Herstellen der Verbindung zum Fremdsystem ein Port benötigt, dann kann dieser Wert in der Methode `customOpen` übergeben werden.

2. Legen Sie im VIP-Administrationsprogramm einen eigenen Pooltyp für die Verbindungen zum Fremdsystem an.

   Geben Sie dabei als *Klassenname* den Namen der neuen Verbindungsklasse an, die Sie von der Basisklasse `PoolConnection` abgeleitet haben. Außerdem können – abhängig vom Fremdsystem – weitere Parameter definiert werden.

3. Legen Sie auf Grundlage dieses Pooltyps einen neuen Pool an. Dieser Pool enthält die Verbindungen zum Fremdsystem und wird vom `PoolManager` verwaltet.

4. Der Agent, der die Verbindung zum Fremdsystem herstellen soll, muss die `PoolManager`-Methode `getConnection` aufrufen, um eine Verbindung aus dem benutzerdefinierten Pool zu erhalten. Der `PoolManager` stellt die Verbindung zum Fremdsystem mithilfe der Methode `customOpen` her. Dabei werden u. a. die Parameter übergeben, die im VIP-Administrationsprogramm für diesen Pool definiert wurden.

9.1 Das Interface `PoolManager`

Der `PoolManager` verwaltet die Pools, die auf Grundlage benutzerdefinierter Pooltypen erstellt wurden. Die Verbindungen in diesen Pools müssen von der Klasse `PoolConnection` abgeleitet sein. Die Pools müssen eindeutige Namen haben, sie werden über das VIP-Administrationsprogramm angelegt und bearbeitet.

Über das Interface `PoolManager` können Sie mit der Methode `getConnection` unter Angabe des Poolnamens eine Verbindung aus dem gewünschten Pool anfordern. Wenn die Verbindung nicht mehr benötigt wird, **muss** sie mit der Methode `freeConnection` an den Pool zurückgegeben werden. Erfolgt die Rückgabe nicht, gilt die Verbindung weiterhin als belegt, auch wenn sie nicht verwendet wird. Damit kann sie nicht für andere Aktionen verwendet werden.

Der Zugriff auf das Interface `PoolManager` erfolgt über `VipRuntime.getPoolManager`.

```
package de.gauss.vip.api.pool;
public interface PoolManager
{
    public Iterator getPoolNames();
    public boolean isExistingPool(String poolName);
    public PoolConnection getConnection(String name);
    public boolean freeConnection(Connection con);
}
```

9.2 Die Basisklasse PoolConnection

Von der abstrakten Klasse PoolConnection müssen Klassen für Verbindungen abgeleitet werden, die auf eigenen Pooltypen basieren. Diese Verbindungen werden dann über den Pool-Mechanismus von VIP ContentManager verwaltet.

```
package de.gauss.vip.api.pool;
public abstract class PoolConnection
{
    abstract public boolean close();
    abstract public boolean isValid();
    abstract public Object execute(Serializable parm1);
    abstract public boolean customOpen(Properties prop);
}
```

Die Methoden close, isValid, execute und customOpen müssen in einer abgeleiteten Klasse ausprogrammiert werden.

close

Diese Methode dient zum Schließen einer benutzerdefinierten Verbindung. Sie wird normalerweise nicht direkt, sondern vom PoolManager aufgerufen. War das Schließen der Verbindung erfolgreich, liefert die Methode true zurück, ansonsten false.

customOpen(Properties prop)

Mit dieser Methode wird eine benutzerdefinierte Verbindung geöffnet. Diese Methode wird ausschließlich vom PoolManager verwendet und darf nicht direkt aufgerufen werden. Wenn im VIP-Administrationsprogramm für den verwendeten Pool zusätzliche Parameter definiert wurden, dann werden diese im Properties-Objekt prop übergeben.

War der Verbindungsaufbau erfolgreich, liefert die Methode true zurück, ansonsten false.

`excute(Serializable parm1)`

Mit dieser Methode werden Befehle über eine geöffnete Verbindung ausgeführt. Mithilfe des Parameters `parm1` werden Parameter für den Befehl wie z. B. eine Befehls-ID übergeben. Sie können beliebige Parameter übergeben. Die Methode liefert ein Objekt zurück, das den Rückgabewert des ausgeführten Befehls repräsentiert.

`isValid`

Mit dieser Methode kann festgestellt werden, ob die entsprechende Verbindung noch geöffnet und gültig ist. Ist dies der Fall, wird `true` zurückgegeben, ansonsten `false`.

KAPITEL 10

Systemverwaltung

In der Ansicht *Systemverwaltung* des VIP-Administrationsprogramms (Admin-Clients) können Sie die Protokolle und Berichte der VIP-CM-Server einsehen sowie Tracing-Funktionen nutzen. Diese Funktionen werden vor allem für die Fehlersuche benötigt.

Der Zugriff auf die Systemverwaltung ist auch über das VIP Java API möglich. Zu diesem Zweck steht das Interface `SystemHandler` zur Verfügung. Es bietet folgende Funktionen:

- Ermitteln der Benutzer, die am VIP-CM-System angemeldet sind (siehe 10.1 “Angemeldete Benutzer ermitteln” auf Seite 232)
- Auslesen der Protokolle der VIP-CM-Server (siehe 10.2 “Protokolle der VIP-CM-Server” auf Seite 232)
- Zugriff auf die Berichte der VIP-CM-Server (siehe 10.3 “Berichte der VIP-CM-Server” auf Seite 234)
- Nutzen von Tracing-Funktionen (siehe 10.4 “Tracing-Funktionen nutzen” auf Seite 235)

Hinweise:

Zugriff auf weitere Funktionen der Systemverwaltung wie Setzen von Runlevels erhalten Sie über das `AdminHandler`-Interface, siehe Kapitel 4 “Administrationsinterface”.

Der Zugriff auf die Funktionen der Systemverwaltung erfordert das Administrationsrecht “Zugriff auf Systemverwaltung”. Weitere Informationen zu den Administrationsrechten finden Sie im VIP ContentManager-Administratorhandbuch.

Eine Instanz des `SystemHandler`-Interfaces erhalten Sie über `VipRuntime.getSystemHandler`.

Eine kurze Beschreibung der einzelnen Methoden des `SystemHandler`-Interfaces wird in den folgenden Abschnitten gegeben. Soweit nicht anders angegeben, befinden sich alle Klassen und Interfaces im Package `de.gauss.vip.api.admin`.

```
package de.gauss.vip.api.admin
public interface SystemHandler
{
  // --- Reading the list of logged-in users
      public List getActiveUsers(ContextId cid, Server server)
              throws VipApiException;

  // --- Reading and deleting server logs
      public String[] getLogFile(ContextId cid, Server server, String
                                 filename, int noOfLinesToSkip)
                                 throws VipApiException;
      public List getLogFiles(ContextId cid, Server server)
              throws VipApiException;
      public int getLogFileNumOfLines(ContextId cid, Server server,
                                      String filename)
                                      throws VipApiException;
      public void deleteLogFiles(ContextId cid, List files2delete,
                                 Server server) throws VipApiException;

  // --- Reading server reports
      public List getCoreReport(ContextId cid, Server server, String
                                object) throws VipApiException;
```

```
  // --- Starting tracing, reading and deleting trace files
      public void setGlobalTraceFilter(ContextId cid, Server server,
                                  List filter) throws VipApiException;
      public void setClassTraceFilter(ContextId cid, Server server,
                                  Map classFilterSettings)
                                  throws VipApiException;
      public void setAcceptClassTraceFilter(ContextId cid, Server
                                  server, boolean active)
                                  throws VipApiException;
      public String[] getTraceFile(ContextId cid, Server server,
                                  String filename, int noOfLinesToSkip)
                                  throws VipApiException;
      public List getTraceFiles(ContextId cid, Server server)
              throws VipApiException;
      public int getTraceFileNumOfLines(ContextId cid, Server server,
                                  String filename)
                                  throws VipApiException;
      public List getGlobalTraceFilter (ContextId cid, Server server)
              throws VipApiException;
      public Map getClassTraceFilter(ContextId cid, Server server)
              throws VipApiException;
      public boolean isClassTraceFilterActive(ContextId cid, Server
                                  server)
                                  throws VipApiException;
      public void deleteTraceFiles(ContextId cid, List files2delete,
                                  Server server) throws VipApiException;
}
```

10.1 Angemeldete Benutzer ermitteln

Die Systemverwaltung liefert einen genauen Überblick über die Benutzer, die derzeit am VIP-CM-System angemeldet sind. Dazu dient die Methode `getActiveUsers(ContextId cid, Server server)`. Wird bei der Abfrage ein Servername angeben, werden nur die an diesem VIP-CM-Server angemeldeten Benutzer ausgegeben. Wenn als Servername `null` angeben wird, werden alle am VIP-CM-System angemeldeten Benutzer aufgelistet.

> **Hinweis:** Über das `AdminHandler`-Interface können Principals angelegt und bearbeitet sowie deren Profile ermittelt werden, siehe Kapitel 4 “Administrationsinterface”.

10.2 Protokolle der VIP-CM-Server

Jeder VIP-CM-Server legt mehrere Protokolle an. Diese Protokolle enthalten Informations- und Fehlermeldungen der Server. Sie sind in den Verzeichnissen **\log\, \log\deployment** sowie **\log\contentminer** des VIP-Installationsverzeichnisses auf dem Rechner gespeichert, auf dem der entsprechende VIP-CM-Server installiert ist. Wie die Berichte können auch die Protokolle zur Problemanalyse und -behebung verwendet werden. Die einzelnen Protokolle können über das VIP-Administrationsprogramm eingesehen werden (siehe VIP ContentManager-Administratorhandbuch).

Der Zugriff auf die Protokolle kann auch über das VIP Java API erfolgen. Zu diesem Zweck stellt das `SystemHandler`-Interface entsprechende Methoden zur Verfügung. So erhalten Sie mit der Methode `public List getLogFiles(ContextId cid, Server server)` unter Angabe des Servernamens eine Liste der zu diesem Server gehörenden Protokolldateien. Den Inhalt einer einzelnen Protokolldatei erhalten Sie mithilfe der Methode `public String getLogFile(ContextId, Server, String, int)` unter Angabe des Dateinamens und des Servers.

Das folgende Beispiel ermittelt zuerst die Liste mit den Namen der Protokolldateien und ruft dann den Inhalt der einzelnen Protokolldateien ab.

```
// get SystemHandler
SystemHandler sysHandler = VipRuntime.getSystemHandler();

// load list with log file names of the current server
Server currentServer = VipRuntime.getCurrentServer();
List logFiles = sysHandler.getLogFiles(cid, currentServer );

//iterate through the result
Iterator logFileIter = logFiles.iterator();
while ( logFileIter.hasNext() )
{
      String logFileName = (String) logFileIter.next();
      //get logs
      int logFileNumOfLines = sysHandler.getLogFileNumOfLines(cid,
                                          currentServer, logFileName );
      String[] logFileContents = new String[logFileNumOfLines];
      logFileContents = sysHandler.getLogFile(cid, currentServer,
                                             logFileName, 0);
...
}
```

Hinweis: Analog können auch die Trace-Protokolle eines VIP-CM-Servers gelesen werden.

10.3 Berichte der VIP-CM-Server

Für die verschiedenen Softwarekomponenten der VIP-CM-Server werden automatisch Berichte erstellt, die die Aktionen im VIP-CM-System protokollieren. Abhängig vom Servertyp und der Konfiguration des Servers werden verschiedene Berichte erstellt – so sind z.B. Deploymentberichte nur für Server verfügbar, auf denen Deploymentsysteme eingerichtet sind. Die verschiedenen Berichte können über das VIP-Administrationsprogramm eingesehen werden (siehe VIP ContentManager-Administratorhandbuch).

Das `SystemHandler`-Interface stellt die Methode public `List getCoreReport(ContextId, Server, String)` zur Verfügung, mit der ein bestimmter Bericht von einem Server angefordert wird. Der Name des Berichts wird als String übergeben. Die zurückgegebene Liste enthält die einzelnen Zeilen des Berichts als String-Objekte. Wird statt des Berichtnamens 0 übergeben, liefert die Methode alle auf diesem Server verfügbaren Berichtobjekte als Strings zurück.

Das folgende Beispiel ermittelt zuerst die Liste mit den Namen der Serverberichte und holt dann den Inhalt der Berichte ab.

```
//get SystemHandler
Server currentServer = VipRuntime.getCurrentServer();
SystemHandler sysHandler = VipRuntime.getSystemHandler();

// load list with report names of the current server
List repObjects = sysHandler.getCoreReport(cid, currentServer, null);

//iterate through the result in order to get the reports
Iterator repIter = repObjects.iterator();
while ( repIter.hasNext() )
{
      String report = (String) repIter.next();
      List reportContents = sysHandler.getCoreReport(cid, currentServer,
                                                             report);
.....
}
```

10.4 Tracing-Funktionen nutzen

Ein weiteres Mittel bei der Fehlersuche stellt das Tracing dar. Mithilfe der Tracing-Funktionen von VIP ContentManager können Sie die laufenden Aktionen und Meldungen eines VIP-CM-Servers in einem Protokoll mitschreiben lassen. Außerdem ist es möglich, über klassenspezifische Filter einzelne Klassen zu tracen. Das Tracing wird wie die anderen Funktionen der Systemverwaltung über das VIP-Administrationsprogramm gesteuert.

Für den Zugriff auf die Tracing-Funktionen über das VIP Java API steht das Interface `TraceFilter` zur Verfügung.

```
public interface TraceFilter
{
      public static final TraceFilter METHOD_ENTER;
      public static final TraceFilter METHOD_LEAVE;
      public static final TraceFilter FATAL_ERROR;
      public static final TraceFilter ERROR;
      public static final TraceFilter WARNING;
      public static final TraceFilter INFO_MESSAGE;

      public String getName();
      public int getId();
      public String getDescription( Locale loc );
}
```

Für das Tracing eines VIP-CM-Servers bzw. einer Klasse kann genau festgelegt werden, welche Ereignisse im Tracing protokolliert werden. Im VIP Java API erfolgt dies über Konstanten des Interfaces `TraceFilter`. Die folgende Tabelle erläutert die Bedeutung der einzelnen Konstanten.

Tabelle 15 – Konstanten für Tracing-Einstellungen (Interface `TraceFilter`)

Konstante	Funktion
`METHOD_ENTER`	Protokollieren der Einsprünge in die Methoden der Klassen, die im VIP-CM-System genutzt werden
`METHOD_LEAVE`	Protokollieren des Verlassens einer Methode
`FATAL_ERROR`	Protokollieren von Fehlern mit dem Level “Schwerwiegend”, die z.B. dazu führen, dass der entsprechende Server nicht mehr lauffähig ist
`ERROR`	Protokollieren von Fehlern mit dem Level “Fehler”, die bei Aktionen im VIP-CM-System auftreten. Fehler mit diesem Level führen in der Regel nicht dazu, dass ein Server nicht mehr lauffähig ist.
`WARNING`	Protokollieren von Fehlern mit dem Level “Warnung”
`INFO_MESSAGE`	Protokollieren von Informationsmeldungen

Das Tracing kann global für den gesamten VIP-CM-Server erfolgen oder auf eine bzw. mehrere Klassen beschränkt werden (klassenspezifisches Tracing).

Das Tracing ist standardmäßig ausgeschaltet. Um das globale Tracing zu aktivieren, wird die Methode `setGlobalTraceFilter( ContextId, Server, List)` verwendet. Für das klassenspezifische Tracing steht die Methode `setClassTraceFilter(ContextIdcid, Serverserver, Map classFilterSettings)` zur Verfügung. Die jeweiligen Tracing-Einstellungen werden in Form einer Liste mit `TraceFilter`-Konstanten übergeben.

Das folgende Beispiel aktiviert ein globales Tracing, in dem das Einspringen und Verlassen von Methoden sowie schwerwiegende Fehler protokolliert werden.

```
// get SystemHandler
Server currentServer = VipRuntime.getCurrentServer();
SystemHandler sysHandler = VipRuntime.getSystemHandler();

// create list with trace settings
List traceFilter = new ArrayList();
traceFilter.add(TraceFilter.METHOD_ENTER);
traceFilter.add(TraceFilter.METHOD_LEAVE);
traceFilter.add(TraceFilter.PANIC);

// activate global tracing for the current server
sysHandler.setGlobalTraceFilter(cid, currentServer, traceFilter);
```

Um den Inhalt des Trace-Protokolls auszulesen, verwenden Sie die Methode `getTraceFile(ContextId cid, Server server, String filename, int noOfLinesToSkip)`. Um die Trace-Einstellungen für einen Server komplett zu löschen und damit das Tracing zu deaktivieren, ist anstelle der Liste in der Methode `setGlobalTraceFilter (ContextId, Server, List.)` `null` zu übergeben.

KAPITEL 11

Anwendungsbeispiele

In diesem Kapitel werden einige einfache Anwendungsbeispiele vorgestellt, die verdeutlichen sollen, wie Server-Agenten entwickelt werden können.

- Im Abschnitt 11.1 “Basisklasse `ExampleAgent`” auf Seite 241 wird eine abstrakte Agentenklasse eingeführt, die notwendige Eigenschaften schon implementiert und die Lokalisierung von Texten demonstriert.
- Der Agent “Example1” zeigt, wie ein Agent aussehen kann, der sich auf das Ereignis “Vorlegen zur Qualitätssicherung” registriert.

  Siehe Abschnitt 11.2 “Protokollieren aller vorgelegten Objekte” auf Seite 243
- Der Agent “Example2” ist ein aktiver Agent, der – nachdem eine Website betriebsbereit ist – alle VIP-Objekte der Website vorlegt, falls diese noch nicht vorgelegt wurden. Dieses Beispiel zeigt, wie eine Registrierung auf Runlevel-Ereignisse realisiert werden kann.

  Siehe Abschnitt 11.3 “Automatisches Vorlegen” auf Seite 247
- Der Agent “Example3” demonstriert den Veto-Mechanismus. Die Benutzer einer bestimmte Gruppe dürfen nur eine vordefinierte Vorlage verwenden, der Agent prüft also, ob diese Regelung eingehalten wurde und legt gegebenenfalls ein Veto gegen die Vorlagenänderung ein.

  Siehe Abschnitt 11.4 “Veto gegen Metadatenänderung” auf Seite 251

- Der Agent “Example4” protokolliert alle Ereignisse, die in einer bestimmten Website auftreten.

  Siehe Abschnitt 11.5 “Event-Protokoll” auf Seite 254

- Der “Java API Test Deployment Agent” veranschaulicht die Registrierung auf Deployment-Ereignisse.

  Siehe Abschnitt 11.6 “Beobachten von Deployment-Ereignissen” auf Seite 257

Die hier vorgestellten Beispiele dienen primär dazu, die Programmierung von Server-Agenten zu verdeutlichen. Der Einfachheit halber wurden bestimmte Funktionen nicht bzw. nicht vollständig implementiert, etwa in den Bereichen Fehlerbehandlung und -protokollierung. Unter realen Bedingungen müsste dies über die entsprechenden Interfaces erfolgen.

> **Hinweis:** Die JAVA-Dateien der Server-Agenten befinden sich im Verzeichnis **\examples\serveragents** des VIP-Installationsverzeichnisses.

11.1 Basisklasse ExampleAgent

Dieses Beispiel implementiert eine Basisklasse, von der die nachfolgenden Beispiel-Agenten ableiten. In dieser Basisklasse sind die gemeinsamen Eigenschaften des Interfaces `ServerAgent` implementiert (Hersteller, Beschreibung und Version des Agenten sowie die erforderliche Version der VIP CM Suite). Durch die Verwendung der Klasse `UserMessage` kann eine lokalisierte Beschreibung des Agenten geliefert werden.

```
package com.company.vip.api.example;

import java.util.Locale;

import de.gauss.vip.api.ServerAgent;
import de.gauss.vip.api.UserMessage;

// Abstract basic class for deriving agents (implements methods
// from de.gauss.vip.api.ServerAgent that return constant values which
// are set via a constructor).
public abstract class ExampleAgent
    implements ServerAgent
{
    // final variables for holding constant values
    private final UserMessage msgM;
    private final String manufacturerM;
    private final String versionM;
    private final int majorM;
    private final int minorM;

    // the only constructor
    public ExampleAgent(String manufacturerArg, String msgKey,
                        String versionArg, int majorArg,
                        int minorArg)
    {
        manufacturerM = manufacturerArg;
        msgM          = new UserMessage(msgKey);
        versionM      = versionArg;
        majorM        = majorArg;
        minorM        = minorArg;
    }

    /**
     * de.gauss.vip.api.ServerAgent implementation (except start(...)
```

```
     * and stop())
     */
    public String getManufacturer()
    {
        return manufacturerM;
    }

    public String getDescription(Locale locale)
    {
        // The message keys have to be defined in the
        // "UserMessage_<langcode>.properties" file which
        // is loaded via a class loader.
        return msgM.getString(locale);
    }

    public String getVersion()
    {
        return versionM;
    }

    public int getRequiredVIPMajorVersion()
    {
        return majorM;
    }

    public int getRequiredVIPMinorVersion()
    {
        return minorM;
    }
}
```

11.2 Protokollieren aller vorgelegten Objekte

Dieser Agent protokolliert jedes Vorlegen eines VIP-Objekts in einer konfigurierten Website. Dazu registriert sich der Agent in der `start`-Methode auf das entsprechende Ereignis (`OBJECT_SUBMITTED_TO_QA`). Für Demonstrationszwecke wird das Protokoll einfach in die Standardausgabe geschrieben.

Die Konfiguration des Agenten erfolgt im VIP-Administrationsprogramm. Dort müssen die Eigenschaften “website”, “agent-user” und “agent-password” für diesen Agenten eingestellt werden. Wird für die Eigenschaft “agent-user” der Wert “backup” eingestellt, meldet sich der Agent nicht selber am VIP-CM-Server an, sondern verwendet den vordefinierten Backup-Kontext, der es ihm erlaubt, alle Objekte einer Website zu lesen.

Die Konfigurationsdaten des Agenten werden im Konstruktor als `Properties`-Objekt übergeben.

```
package com.company.vip.api.example;

import java.util.ArrayList;
import java.util.List;
import java.util.Properties;

import de.gauss.vip.api.ServerAgent;
import de.gauss.vip.api.VipRuntime;
import de.gauss.vip.api.event.EventDispatcher;
import de.gauss.vip.api.event.EventListener;
import de.gauss.vip.api.event.Event;
import de.gauss.vip.api.object.ObjectHandler;
import de.gauss.vip.api.object.ObjectData;
import de.gauss.vip.api.exception.VipApiException;
import de.gauss.vip.api.lang.ContextId;
import de.gauss.vip.api.lang.ObjectId;
import de.gauss.vip.api.admin.ContextHandler;
import de.gauss.vip.api.admin.Runlevel;

// This agent registers itself for QA submission events for a
// configurable website and prints some values to stdout for any
// submit event it receives.
```

```
final public class Example1
    extends ExampleAgent
    implements EventListener
{
    // holds EventDispatcher (singleton)
    private final static EventDispatcher eventDispatcherM
        = VipRuntime.getEventDispatcher();

    // set from "website" parameter
    private final String websiteNameM;
    // set from "agent-user" parameter
    private final String agentUserM;
    // set from "agent-password" parameter
    private final String agentPasswordM;
    // set from "agent-user" parameter
    private final boolean backupM;
    // holds ContextHandler singleton
    private ContextHandler cxtHandlerM;
    // holds ObjectHandler for website
    private ObjectHandler objHandlerM;

      //Event listener which will do some initialization for the agent
    private final EventListener runlevelEventListener =
         new RunlevelListener();

    // holds either user or backup context (set in start(...))
    private ContextId contextId = null;

    // Constructor reads configuration parameters and performs
    // initialization
    public Example1(Properties config)
        throws VipApiException
    {
        // "EXAMPLE1_DESC" will be localized via
        // "UserMessage_<langcode>.properties" file
        super("Company.Com", "EXAMPLE1_DESC", "1.0", 8, 0);

        websiteNameM   = config.getProperty("website");
        agentUserM     = config.getProperty("agent-user");
        agentPasswordM = config.getProperty("agent-password");
        backupM        = agentUserM.equalsIgnoreCase("backup");
    }

    /**
     * de.gauss.vip.api.ServerAgent implementation that is not covered
     * by ExampleAgent.
     */

    // register this agent instance for website/events,
```

```
// register the runlevelEventListener
public boolean start(String serverType, int majorVersion,
                     int minorVersion, String patchLevel)
{
     System.out.println("Example1 : Starting agent for website " +
                        websiteNameM);

     //register 2 Listeners. The runlevelEventListener performs the
     //login of the user and retrieves the ObjectHandler when the
     //website is ready.
     eventDispatcherM.addListener(null, Event.RUNLEVEL_IS,
                                  runlevelEventListener);
     eventDispatcherM.addListener(websiteNameM,
                                Event.OBJECT_SUBMITTED_TO_QA, this);

    return true; // success
}

// Deregister this instance for all websites/events (log out in
// the case of a non-backup context). Note that there is a race
// condition here: there is a slight chance that this agent will
// receive an event and has been logged out (c.f. below).
public void stop()
{
    eventDispatcherM.removeListener(this);
    eventDispatcherM.removeListener(runlevelEventListener);

    if (!backupM)
    {
        try
        {
            cxtHandlerM.stopContextRefresh(contextId);
            cxtHandlerM.logout(contextId);
        }
        catch (VipApiException vae)
        {
            vae.printStackTrace();
        }
    }
}

/**
 * de.gauss.vip.api.event.EventListener implementation
 */

// receives event and reads some values and prints them to stdout
public void performVipEvent(Event event)
{
    try
```

```
        {
            ObjectId objId = (ObjectId)event.getArgument(Event.ARG_OID);
            // the contextId might have been logged out when getting
            // here (c.f. race condition above)
            ObjectData objData = objHandlerM.get(contextId, objId);
            System.out.print("Submitted '" + objData.getTitle()
                            + "' to QA (objId = " + objId.getId() + ")");
        }
        catch (VipApiException vae)
        {
            vae.printStackTrace();
        }
    }

      //Listener which performs the login and retrieves the ObjectHandler
      //when the runlevel Runlevel.WEBSITE_UP is reached.
    class RunlevelListener implements EventListener
      {
         public void performVipEvent(Event ev)
         {
            Integer level = (Integer) ev.getArgument(Event.ARG_NEW);
            String eventWebsite = (String)
                   ev.getArgument(Event.ARG_WEBSITE);
            if (level.intValue() == Runlevel.WEBSITE_UP && eventWebsite
                         != null && eventWebsite.equals(websiteNameM))
            {
               try
               {
                 cxtHandlerM = VipRuntime.getContextHandler();
                 contextId = cxtHandlerM.login(agentUserM,
                                  agentPasswordM.toCharArray());
                 cxtHandlerM.startContextRefresh(contextId);
                 //Since VIP 8.1.1, the ObjectHandler cannot be
                 //retrieved until the website is in runlevel
                 //Runlevel.WEBSITE_UP
               objHandlerM = VipRuntime.getObjectHandler (websiteNameM);
               }
               catch (VipApiException e)
               {
                  System.err.println("Example1 agent : Error: " + e);
               }
            }
         }
      }

}
```

11.3 Automatisches Vorlegen

Dieser Agent demonstriert die Programmierung von aktiven Agenten. Der Agent wird nicht durch Workflow-Ereignisse gesteuert, sondern wird direkt nach dem Hochfahren der konfigurierten Website aktiv.

Der Zugriff auf die VIP-Objekte der Website ist erst möglich, wenn die Website den Runlevel `WEBSITE_UP` erreicht hat. Der Agent registriert sich daher auf das Ereignis für eine Runlevel-Änderung und prüft, ob für die Website dieser Runlevel erreicht wurde. Erst dann wird er aktiv.

Nach dem Hochfahren der Website sucht der Agent alle VIP-Objekte, die noch niemals zur Qualitätssicherung vorgelegt wurden (dazu verwendet er die Suchfunktionalität des `ObjectHandler`-Interfaces) und legt die gefundenen VIP-Objekte vor.

```
package com.company.vip.api.example;

import java.util.Properties;
import java.util.List;

import de.gauss.vip.lang.filter.Filter;
import de.gauss.vip.lang.filter.AndFilter;
import de.gauss.vip.lang.filter.EqualFilter;
import de.gauss.lang.LongValue;

import de.gauss.vip.api.event.EventListener;
import de.gauss.vip.api.event.Event;
import de.gauss.vip.api.object.ObjectHandlerUtil;
import de.gauss.vip.api.object.SearchableKeys;
import de.gauss.vip.api.exception.VipApiException;
import de.gauss.vip.api.lang.ObjectState;
import de.gauss.vip.api.admin.Runlevel;

import de.gauss.vip.api.admin.ContextHandler;
import de.gauss.vip.api.lang.ContextId;
import de.gauss.vip.api.object.ObjectHandler;
import de.gauss.vip.api.VipRuntime;
import de.gauss.vip.api.event.EventDispatcher;

// This agent registers itself for RUNLEVEL_IS events for a
// configurable website and synchronously submits any edited object
// that has never been submitted to QA.
```

```
public class Example2
    extends ExampleAgent
    implements EventListener
{
    // holds EventDispatcher (singleton)
    private final static EventDispatcher eventDispatcherM
        = VipRuntime.getEventDispatcher();

    // set from "website" parameter
    private final String websiteNameM;
    // set from "agent-user" parameter
    private final String agentUserM;
    // set from "agent-password" parameter
    private final String agentPasswordM;
    private Filter filterM;

    // Constructor reads configuration parameters and prepares the
    // filter.
    public Example2(Properties config)
        throws VipApiException
    {
        // "EXAMPLE2_DESC" will be localized via
        // "UserMessage_<langcode>.properties" file
        super("Company.Com", "EXAMPLE2_DESC", "1.1", 8, 0);

        websiteNameM   = config.getProperty("website");
        agentUserM     = config.getProperty("agent-user");
        agentPasswordM = config.getProperty("agent-password");

    }

    /**
     * de.gauss.vip.api.ServerAgent implementation that is not covered
     * by ExampleAgent
     */

    // register this instance for events
    public boolean start(String serverType, int majorVersion,
                         int minorVersion, String patchLevel)
    {
        eventDispatcherM.addListener(null, Event.RUNLEVEL_IS, this);
        return true; // success
    }

    // deregister this instance for all websites/events
    public void stop()
    {
        eventDispatcherM.removeListener(this);
    }
```

```
/**
 * de.gauss.vip.api.event.EventListener implementation
 */

// If the configured website has just been started, this method
// submits any edited object that has never been submitted to QA (the
// submisson is done synchronously --- so event propagation will
// pause until the submission completes). For 'real' agents this
// should be changed to asynchronous submission, but special care
// would have to be taken for context refresh and subsequent
// events, e.g. WEBSITE_INACCESSIBLE).
public void performVipEvent(Event event)
{
    Integer level  = (Integer)event.getArgument(Event.ARG_NEW);
    String website = (String)event.getArgument(Event.ARG_WEBSITE);

    if (level.intValue() == Runlevel.WEBSITE_UP
        && website != null && website.equals(websiteNameM))
    {
         ContextHandler cxtHandler = null;
         ContextId contextId       = null;
        try
        {
            // construct filter
            ObjectState objState =
                 ObjectHandlerUtil.getObjectState(websiteNameM,
                                              ObjectState.EDITED);
            Filter objStateFilter =
                 new EqualFilter(SearchableKeys.STATE,objState);
            Filter versionFilter  =
                 new AndFilter(new EqualFilter
                                  (SearchableKeys.VERSION_MINOR,
                                      new LongValue(0)),
                              new EqualFilter(SearchableKeys.
                                    VERSION_MAJOR,
                                     new LongValue(0)));

           filterM = new AndFilter(objStateFilter, versionFilter);
           ObjectHandler objHandler
               = VipRuntime.getObjectHandler(websiteNameM);
           cxtHandler = VipRuntime.getContextHandler();
           contextId  = cxtHandler.login(agentUserM,
                                    agentPasswordM.toCharArray());

           // find objects and sort the result so that parents
           // will be displayed before their children
           List objects = objHandler.sortParentsFirst(contextId,
                                      objHandler.filter(contextId,
```

```
                                                                    filterM,
                                                                 null, null,
                                                                    0, -1));
                // submit (no pending release date and no e-mail)
                objHandler.submit(contextId, objects, null,
                                  "agent's remark", null);
            }
            catch (VipApiException vae)
            {
                vae.printStackTrace();
            }
            finally
            {
                try
                {
                    if (cxtHandler != null && contextId != null)
                        cxtHandler.logout(contextId);
                }
                catch (VipApiException vax)
                {
                    vax.printStackTrace();
                }
            }
        }
    }
}
```

11.4 Veto gegen Metadatenänderung

Dieser Agent demonstriert die Registrierung und Verarbeitung von Vorbereitungsereignissen (`PrepareEvents`). Der Agent verbietet es, den Titel von VIP-Objekten zu ändern oder eine andere als die konfigurierte Vorlage zu verwenden. Wurde der Titel geändert oder eine Vorlage ausgewählt, die nicht der konfigurierten Vorlage des Agenten entspricht, wird vom Agenten eine `VetoException` geworfen. Dadurch wird die Aktion unterbunden. Die verwendete lokalisierte Nachricht wird dann z. B. im CMS-Client des Benutzers angezeigt.

In diesem Beispiel wird eine innere Klasse verwendet, um das `PrepareEventListener`-Interface zu implementieren.

```
package com.company.vip.api.example;

import java.util.Properties;

import de.gauss.vip.api.VipRuntime;
import de.gauss.vip.api.UserMessage;
import de.gauss.vip.api.event.PrepareEventListener;
import de.gauss.vip.api.event.EventDispatcher;
import de.gauss.vip.api.event.PrepareEvent;
import de.gauss.vip.api.event.Event;
import de.gauss.vip.api.event.VetoException;
import de.gauss.vip.api.lang.ObjectId;
import de.gauss.vip.api.object.FieldNames;
import de.gauss.vip.repository.RepositoryEntry;

// This agent registers an inner class instance for
// PrepareEvent.PREPARE_OBJECT_CHANGE_METADATA events for a
// configurable website and prevents title changes and template
// assignments via veto exceptions.
public class Example3
    extends ExampleAgent
{
    // These keys will be localized via
    // "UserMessage_<langcode>.properties" file
    private final static String templateVetoKeyM
        = "VETO_TEMPLATE_FOR_{0}_SHOULD_BE_{1}";

    private final static String titleVetoKeyM
        = "VETO_TITLE_FOR_{0}_NOT_CHANGEABLE";
```

```
    // keep the EventDispatcher singleton
    private final static EventDispatcher eventDispatcherM
        = VipRuntime.getEventDispatcher();

    // blank final fields

    // set from "website" config parameter
    private final String websiteNameM;
    // set from "template" config parameter
    private final ObjectId templateM;

    // Construct the PrepareChangeListener (inner class; c.f. below)
    // that will receive events after it has been registered (from
    // within start(...))
    private final PrepareEventListener eventListenerM
        = new PrepareChangeListener();

    // constructor reads configuration parameters
    public Example3(Properties config)
    {
        super("Company.Com", "EXAMPLE3_DESC", "1.1", 8, 0);

        websiteNameM = config.getProperty("website");
        templateM    = new ObjectId(config.getProperty("template"));
    }

    /**
     * de.gauss.vip.api.ServerAgent implementation that is not covered
     * by ExampleAgent.
     */

    // register the PrepareChangeListener instance for website/event
    public boolean start(String serverType, int majorVersion,
                         int minorVersion, String patchLevel)
    {
        eventDispatcherM.addListener(websiteNameM,

PrepareEvent.PREPARE_OBJECT_CHANGE_METADATA,
                                      eventListenerM);
        return true; // this agent wants to be stopped explicitly
    }

    // deregister the PrepareChangeListener instance for all
    // websites/events
    public void stop()
    {
        eventDispatcherM.removeListener(eventListenerM);
    }
```

```
    // non-static inner class -- will receive events and accesses some
    // fields from enclosing class/instance
    private class PrepareChangeListener
        implements PrepareEventListener
    {
        // throw veto exception, if wrong template assignment or if
        // title has been changed
        public void performVipPrepareEvent(PrepareEvent event)
            throws VetoException
        {
            ObjectId objId
                = (ObjectId)event.getArgument(Event.ARG_OID);
            RepositoryEntry changed
                = (RepositoryEntry)event.getArgument(Event.ARG_NEW);

            if (changed.containsKey(FieldNames.TEMPLATE))
            {
                ObjectId newTemplate
                    = (ObjectId)changed.get(FieldNames.TEMPLATE);
                if (newTemplate != null &&
                                  !newTemplate.equals(templateM))
                    throw new VetoException(new
                                          UserMessage(templateVetoKeyM,
                                                      objId.getId(),
                                                    templateM.getId()));
            }
            if (changed.containsKey(FieldNames.TITLE))
                throw new VetoException(new UserMessage(titleVetoKeyM,
                                                        objId.getId()));
        }
    }
}
```

11.5 Event-Protokoll

Dieser Agent protokolliert alle Ereignisse, die für die eingestellte Website auftreten. Die Ereignisse und die zugehörigen Argumente werden in die Standardausgabe geschrieben. Es werden keine Vorbereitungsereignisse verarbeitet. Sowohl die Website als auch die zu protokollierenden Ereignisse werden in der Konfiguration des Agenten eingestellt. In der Eigenschaft “event-ids” (in der Konfiguration des Agenten) werden die Ereignisse als komma- bzw. leerzeichengetrennte Liste von Zahlen angegeben. Um alle Ereignisse zu berücksichtigen, wird dort “*” eingestellt.

```
package com.company.vip.api.example;

import java.util.Properties;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.StringTokenizer;

import de.gauss.vip.api.VipRuntime;
import de.gauss.vip.api.event.EventDispatcher;
import de.gauss.vip.api.event.EventListener;
import de.gauss.vip.api.event.Event;

// This agent registers itself for a configurable website and set of
// events and prints some values to stdout for any events it receives.
public class Example4
    extends ExampleAgent
    implements EventListener
{
    // "*" in property "event-ids" will result in registering for all
    // of these events
    private static final int[] ALL_EVENTS =
    {
        Event.OBJECT_METADATA_CHANGED,
        Event.OBJECT_CREATED,
        Event.OBJECT_DELETED,
        Event.OBJECT_DESTROYED,
        Event.OBJECT_CHECKED_OUT,
        Event.OBJECT_CHECKOUT_UNDONE,
        Event.OBJECT_CHECKED_IN,
        Event.OBJECT_SUBMITTED_TO_QA,
        Event.OBJECT_SUBMITTED_TO_PRODUCTION,
        Event.OBJECT_REJECTED_TO_EDIT,
```

```
        Event.OBJECT_ACL_CHANGED,
        Event.OBJECT_SUBMIT_AT_TO_PRODUCTION,
        Event.WEBSITE_NEW,
        Event.WEBSITE_DELETED,
        Event.OBJECT_EXPIRED,
        Event.USER_LOGIN,
        Event.USER_LOGOUT,
        Event.RUNLEVEL_INCREASES,
        Event.RUNLEVEL_DECREASES,
        Event.RUNLEVEL_IS
    };

    // get the EventDispatcher (singleton)
    private final static EventDispatcher eventDispatcherM
        = VipRuntime.getEventDispatcher();

    // blank final fields

    // set from "website" config parameter
    private final String websiteNameM;
    private final List useEventIdsM = new ArrayList();

    // Constructor reads config parameters and constructs set of
    // events to register for
    public Example4(Properties config)
        throws NumberFormatException
    {
        // "EXAMPLE4_DESC" will be localized via
        // "UserMessage_<langcode>.properties" file
        super("Company.com", "EXAMPLE4_DESC", "2.3", 8, 0);

        websiteNameM = config.getProperty("website");
        String eventIds = config.getProperty("event-ids");

        // process config parameter "event-ids" --- "*" will add all
        // events from ALL_EVENTS[] any other value will be interpreted
        // as a comma/space-separated list of integer values denoting the
        // event numbers to register for
        if (eventIds.equals("*"))
          for (int i = 0; i < ALL_EVENTS.length; i++)
                useEventIdsM.add(new Integer(ALL_EVENTS[i]));
        else
         for (StringTokenizer stok = new StringTokenizer(eventIds, ", ");
                stok.hasMoreTokens();)
                useEventIdsM.add(new Integer((String)stok.nextToken()));
    }

    // auxiliary method
    private static boolean isSystemEvent(int id)
```

```
    {
        return (id == Event.USER_LOGIN
                || id == Event.USER_LOGOUT
                || id == Event.WEBSITE_NEW
                || id == Event.WEBSITE_DELETED
                || id == Event.RUNLEVEL_INCREASES
                || id == Event.RUNLEVEL_DECREASES
                || id == Event.RUNLEVEL_IS);
    }

    /**
     * de.gauss.vip.api.ServerAgent implementation that is not covered
     * by ExampleAgent.
     */

    // register this agent instance for website/events
    public boolean start(String serverType, int majorVersion,
                         int minorVersion, String patchLevel)
    {
        for (Iterator i = useEventIdsM.iterator(); i.hasNext();)
        {
            int id = ((Integer)i.next()).intValue();
            if (isSystemEvent(id))
                eventDispatcherM.addListener(id, this);
            else
                eventDispatcherM.addListener(websiteNameM, id, this);
        }
        return true;
    }

    // deregister this instance for all websites/events
    public void stop()
    {
        eventDispatcherM.removeListener(this);
    }

    /**
     * de.gauss.vip.api.event.EventListener implementation
     */

    // receives event and reads some values and prints them to stdout
    public void performVipEvent(Event event)
    {
        System.out.println("Event id=" + event.getId()
                               + " args=" + event.getArguments());
    }
}
```

11.6 Beobachten von Deployment-Ereignissen

Dieses Beispiel zeigt eine einfache Implementierung eines Agenten, der alle Deployment-Ereignisse beobachtet und auf die Konsole schreibt.

```
import java.util.LinkedList;
import java.util.Iterator;
import java.util.List;
import de.gauss.vip.api.event.DeploymentEvent;
import de.gauss.vip.api.event.DeploymentEventDispatcher;
import de.gauss.vip.api.event.DeploymentEventListener;
import de.gauss.vip.api.VipRuntime;
import de.gauss.lang.StringValue;
import de.gauss.vip.api.ServerAgent;
import java.util.Locale;
import de.gauss.vip.api.lang.ObjectId;
import de.gauss.vip.config.KeyNotFoundException;

/**
 * Title:
 * Description:  A small agent for deployment events.<br>
 *               Starts a thread printing out deployment events.
 * Copyright:    Copyright (c) 2002
 * Company:
 * @author
 * @version 1.0
 */
public class DplEventTester
    extends ExampleAgent
    implements DeploymentEventListener, ServerAgent, Runnable
{
    static int counterM = 0;
    private Object mutexM = new Object();
    private String nameM = "DeplEventTester_";
    private volatile boolean stopM = false;
    private LinkedList todoM;
    private de.gauss.vip.config.PropertyMap pmM;

    public DplEventTester(de.gauss.vip.config.PropertyMap pm)
    {
        super("Gauss Interprise AG", "Java API Test Deployment Agent",
              "0.2", 8, 0);
        pmM = pm;
        synchronized (mutexM)
```

```
        {
            counterM++;
            nameM = nameM + counterM;
            todoM = new LinkedList();
        }
    }

    // --- interface ServerAgent. Caution: some methods are inherited
    // from the super class.

    /**
     * Starts a thread for printing out events.
     *
     * @param serverType the type of the VIP CM server that will execute
     * this agent.
     * @param majorVersion the major release number of
     * VIP ContentManager.
     * @param manorVersion the minor release number of
     * VIP ContentManager.
     * @param patchLevel the micro (patch level) release number of the
     * VIP ContentManager.
     * @return <tt>true</tt> because the agent wants {@link #stop()} to
     * be called.
     */
    public boolean start(String serverType, int majorVersion,
                         int minorVersion, String patchLevel)
    {
        try
        {
            stopM = false;
            Thread thread = new Thread(this, nameM);
            //Start the thread's run method.
            thread.start();
            return true;
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            return false;
        }
    }

    /**
     * Stops the thread the agent has started.
     */
    public void stop()
    {
        try
        {
```

```
            synchronized(mutexM)
            {
                stopM = true;
                mutexM.notify();
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }

    // --- interface EventListener
    /**
     * Adds the given event to the internal todo queue.
     *
     * @param event Event to be handled.
     */
    public void performVipDeploymentEvent(DeploymentEvent event)
    {
        synchronized (mutexM)
        {
            todoM.addLast(event);
            mutexM.notify();
        }
    }


   // --- interface Runnable
   /**
    * Registers for deployment events.
    * Waits for events to be performed (in a loop).
    * Leaves the loop as soon as {@link #stop()} has been invoked.
    * Deregisters for deplyoment events before returning.
    */
    public  void run()
    {
        DeploymentEvent event = null;

        try
        {
            System.out.println( "Thread " + nameM
                               + " has configuration " + reportPM(pmM));
            VipRuntime.getDeploymentEventDispatcher().addListener(
                       DeploymentEvent.FILE_CHANGED, this);
            VipRuntime.getDeploymentEventDispatcher().addListener(
                       DeploymentEvent.FILE_DELETED, this);
            VipRuntime.getDeploymentEventDispatcher().addListener(
                       DeploymentEvent.DEPLOYMENT_SYSTEM_CREATED, this);
```

```
        VipRuntime.getDeploymentEventDispatcher().addListener(
                   DeploymentEvent.DEPLOYMENT_SYSTEM_DELETED, this);
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
    do
    {
        try
        {
            synchronized(mutexM)
            {
                if (!stopM)
                {
                    if (todoM.isEmpty())
                    {
                        try
                        {
                            mutexM.wait(300000);
                        }
                        catch(InterruptedException ex)
                        {
                            //Don't do anything!
                        }
                    }
                    else
                    {
                       event = (DeploymentEvent)todoM.removeFirst();
                    }
                }
            }
            if (event != null)
                printEvent(event);
            event = null;
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
        }
    } while(!stopM);
    try
    {
        VipRuntime.getDeploymentEventDispatcher().removeListener(
                   DeploymentEvent.FILE_CHANGED, this);
        VipRuntime.getDeploymentEventDispatcher().removeListener(
                   DeploymentEvent.FILE_DELETED, this);
        VipRuntime.getDeploymentEventDispatcher().removeListener(
                   DeploymentEvent.DEPLOYMENT_SYSTEM_CREATED, this);
```

```
            VipRuntime.getDeploymentEventDispatcher().removeListener(
                       DeploymentEvent.DEPLOYMENT_SYSTEM_DELETED, this);

            System.out.println( "Thread " + nameM
                                + " will be stopped immediately.");
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }

    //Tools
    /**
     * Prints the given event.
     * @param event Event to be printed.
     */
    private void printEvent(DeploymentEvent event)
    {
        switch (event.getId())
        {
            case DeploymentEvent.FILE_CHANGED:
            case DeploymentEvent.FILE_DELETED:
                System.out.println(
                    "Thread   " + nameM + "\n" +
                    "* Event: " + event.getId()+
                    "("+reportEvent(event.getId())+")\n"+
                    "* Dpl:   " + (StringValue)event.getConstraint() +
                    "\n" + "* Oid:   " +
                   (ObjectId)event.getArgument(DeploymentEvent.ARG_OID)+
                    "\n" +
                    "* Path:  " + (StringValue)event.getArgument(
                                   DeploymentEvent.ARG_PATH)+"\n" +
                    "* Url :  " + (StringValue)event.getArgument(
                                   DeploymentEvent.ARG_URL));
                break;
            case DeploymentEvent.DEPLOYMENT_SYSTEM_CREATED:
            case DeploymentEvent.DEPLOYMENT_SYSTEM_DELETED:
                System.out.println(
                    "Thread   " + nameM + "\n" +
                    "* Event: " + event.getId()+
                    "("+reportEvent(event.getId())+")\n"+
                    "* Dpl:   " + (StringValue)event.getArgument(
                                DeploymentEvent.ARG_DEPLOYMENT_SYSTEM));
                break;
            default:
                System.out.println(
                    "Thread " + nameM + "\n" +
                    "* Event: " + event.getId()+
```

```
                "("+reportEvent(event.getId())+")");
            break;
        }
    }

    /**
     * Converts the given event id to a <tt>String</tt>.
     * @param id Event id.
     * @return String describing the event.
     */
    private String reportEvent(int id)
    {
        switch (id)
        {
            case DeploymentEvent.FILE_CHANGED:
                return "FILE_CHANGED";
            case DeploymentEvent.FILE_DELETED:
                return "FILE_DELETED";
            case DeploymentEvent.DEPLOYMENT_SYSTEM_CREATED:
                return "DEPLOYMENT_SYSTEM_CREATED";
            case DeploymentEvent.DEPLOYMENT_SYSTEM_DELETED:
                return "DEPLOYMENT_SYSTEM_DELETED";
            default:
                return "unknown";
        }
    }

    /**
     * Converts the given property map to a <tt>String</tt>.
     * @param pm Property map containing configuration data.
     * @return String describing the contents of the given property map.
     */
    private String reportPM(de.gauss.vip.config.PropertyMap pm)
    {
        if (pm != null)
        {
            List l = pm.getChildrenKeys();
            if (l != null)
            {
                StringBuffer ret = new StringBuffer();
                Iterator it = l.iterator();
                while (it.hasNext())
                {
                    String key = (String)it.next();
                    ret = ret.append(key);
                    ret = ret.append(":");
                    try
                    {
                        String value = pm.getPropertyValue(key);
```

```
                        ret = ret.append(value);
                        ret = ret.append(" ");
                    }
                    catch (KeyNotFoundException notFound)
                    {
                      ret = ret.append("error! ");
                    }
                }
                return ret.toString();
            }
        }
        return "-";
     }
}
```

Glossar

ACL – Access Control List, siehe *Zugriffssteuerungsliste*

Agent – Ein zusätzliches Funktionsmodul auf der Basis des VIP Java API von VIP ContentManager. Durch Agenten wird die Standardfunktionalität der VIP-CM-Server erweitert.

Aktion – Schritt, der zur Verwaltung von Website-Objekten notwendig ist und meistens zur Zustandsänderung des betreffenden Objekts führt

API – Application Programming Interface. VIP ContentManager bietet drei APIs, um auf die Funktionalitäten der VIP-CM-Server zuzugreifen: das VIP Java API, VIP PortalManager und VIP WebServices.

asynchrone Aktion – Eine asynchrone Aktion kehrt praktisch sofort nach dem Aufruf zurück, wartet also nicht, bis die damit verbundenen Aufgaben abgeschlossen sind. Siehe auch *synchrone Aktion*.

Ausleihen – Workflow-Aktion von VIP ContentManager: Der Inhalt eines VIP-Objekts kann erst bearbeitet werden, wenn das Objekt ausgeliehen wurde. Ein ausgeliehenes Objekt ist für die Zugriffe anderer Benutzer gesperrt. Die Änderungen, die am Objekt vorgenommen werden, sind erst nach dem Zurückgeben in der Edit-Sicht verfügbar.

Context-ID – Ein Objekt, das einem Benutzer nach erfolgreicher Anmeldung am VIP-CM-System zugeteilt wird. Eine Context-ID ist immer systemweit eindeutig, sie identifiziert deshalb genau einen Benutzer. Wird eine Context-ID eine gewisse Zeit nicht gebraucht, so verfällt sie.

Datenhaltungssicht – Die Datenhaltungssicht eines Servers bezeichnet die aktuell verfügbaren VIP-Objekte des jeweiligen Servertyps (Edit, QS oder Produktion).

Deployment – Deployment bezeichnet die Verteilung von Daten. Das Deployment von VIP ContentManager übernimmt zwei Hauptaufgaben: erstens die Generierung von Web-Objekten aus den in der Datenbank

gespeicherten VIP-Objekten und die Verteilung der generierten Dateien in die dafür vorgesehenen Verzeichnisse; zweitens die Benachrichtung der VIP-CM-Server bei Änderungen im VIP-CM-System.

DeploymentEvent – Ereignis, das Aktivitäten (auf suchenden und empfangenden VIP-CM-Systemen) bezüglich der Verteilung von Daten zwischen VIP-CM-Servern signalisiert

DeploymentHandler – Instanz, die Deployment-Aktionen aktiv einleitet

Deploymentsystem – Die Deploymentsysteme erzeugen aus den VIP-Objekten Web-Objekte und verteilen die generierten Dateien in die dafür vorgesehenen Verzeichnisse. Von dort aus werden die Dateien über den Einsatz eines HTTP-Servers für die Benutzer sichtbar. Deploymentsysteme können unterschiedliche Typen und Kategorien haben.

Deploymentsystem-Kategorie – Je nach Art der Verarbeitung von Deploymentaufträgen werden Deploymentsysteme verschiedenen Kategorien zugeordnet: *Standard-Deploymentsysteme* legen die erzeugten Web-Objekte in einer hierarchischen Dateistruktur ab und aktualisieren die Web-Objekte automatisch bei Änderung der VIP-Objekte. *Dynamische Deploymentsysteme* generieren die Web-Objekte auf Grundlage benutzerdefinierter Einstellungen und nur dann, wenn das Web-Objekt über den HTTP-Server angefordert wird. Die generierten Dateien werden in einer flachen Dateistruktur abgelegt. Mit *Suchmaschinen-Deploymentsystemen* können Sie die Daten Ihrer Website für den Einsatz einer Suchmaschine aufbereiten. *WebDAV-Deploymentsysteme* bilden die Voraussetzung für den Einsatz von WebDAV-Clients.

Deploymentsystem-Typen – Bei Deploymentsystemen wird entsprechend dem Workflow von VIP ContentManager zwischen den Typen “Edit”, “QS” und “Produktion” unterschieden. Je nach Typ werden unterschiedliche Sichten auf die Daten der Website erzeugt.

Edit-Sicht – In der Edit-Sicht von VIP ContentManager werden die Objekte einer Website angelegt und redaktionell bearbeitet. Hier ist der jeweils aktuelle Bearbeitungsstand zu sehen.

Ereignis – Ereignisse werden durch das VIP-CM-System gefeuert, wenn sich der Zustand eines VIP-Objekts, eines Web-Objekts oder der Systemzustand ändert.

Freigeben – Workflow-Aktion von VIP ContentManager: Die Qualitätssicherung prüft inhaltlich und formal, ob ein vorgelegtes Objekt den Qualitätsstandards des Unternehmens entspricht. Ist dies der Fall, wird das Objekt freigegeben. Durch die Freigabe wird die qualitätsgesicherte Version des Objekts in die Produktionssicht übertragen und damit dem Endbenutzer in der publizierten Website verfügbar gemacht.

Kontext – Kontexte dienen dazu, Benutzeranmeldungen und Transaktionen zu identifizieren. Sie werden über das Interface `ContextHandler` verwaltet. Es wird zwischen Benutzer- und Transaktionskontexten unterschieden.

Navigationstopologie – Die nach Themen angeordnete Struktur von VIP-Objekten innerhalb einer Website (Themenstruktur)

ObjectHandler – Diese Instanz nimmt Aktionen an und manipuliert so die Objekte der Website.

Objektdaten – Die Bestandteile eines VIP-Objekts: *Inhalt*, *Metadaten* und *Protokoll*

Objektkategorie – Zuordnung eines VIP-Objekts zu einer bestimmten Kategorie. Aufgrund dieser Zuordnung hat das VIP-Objekt eine Reihe von zusätzlichen Spezialattributen (Metadaten).

Objekttyp – Bezeichnet die spezifische Art des jeweiligen Objekts, z.B. “HTML-Seite”, “HTML-Vorlage”, “Thema”. Aus dem Objekttyp ergeben sich verschiedene Eigenschaften des VIP-Objekts. Der Objekttyp wird beim Anlegen des Objekts festgelegt und kann bei bestimmten

Objekttypen nachträglich verändert werden. Objekttypen können über das VIP-Administrationsprogramm bearbeitet werden.

Objektzustand – Der Bearbeitungszustand eines VIP-Objekts. Änderungen im Objektzustand werden durch entsprechende Aktionen auf dem VIP-Objekt veranlasst.

Produktionssicht – Die Produktionssicht von VIP ContentManager stellt die freigegebenen Seiten einer Website bereit. Mithilfe eines Webservers kann auf die Seiten über das Internet, Intranet oder Extranet zugegriffen werden.

QS-Sicht – Die QS-Sicht von VIP ContentManager dient der Qualitätssicherung der Objekte und damit der Website-Inhalte. Diese Sicht stellt also die Kontrollinstanz zwischen der Bearbeitung in der Edit-Sicht und der Veröffentlichung in der Produktionssicht dar.

Server-Agent – Java-Implementation auf Basis des VIP Java API. Durch Server-Agenten kann die Funktionalität der VIP-CM-Server ergänzt bzw. erweitert werden.

Server-Event – Ereignis, das einen internen Verarbeitungsschritt im VIP-CM-Server bezüglich des Object-Managements anzeigt

Session – Einheit, die von der JSP-Engine verwaltet wird, damit logisch zusammenhängende Aktionen (aus Sicht der Ressourcen) auch zusammengefasst werden können.

synchrone Aktion – Eine synchrone Aktion kehrt erst zurück, wenn alle damit verbundenen Aufgaben abgeschlossen sind. Siehe auch *asynchrone Aktion*.

Topologie – Die hierarchische Anordnung von VIP-Objekten nach bestimmten Kriterien. Neben der *Navigationstopologie* (Anordnung nach Themen mit ihren jeweiligen Unterobjekten) gibt es auch eine *Vorlagentopologie*.

Transaktion – Durch eine Transaktion können mehrere Aktionen miteinander verbunden werden, sodass alle Änderungen erst wirksam werden, wenn die Transaktion explizit beendet wird.

VIPP – VIP Protocol. Proprietäres Protokoll zum Austausch von Daten zwischen den Komponenten eines VIP-CM-Systems. Für die Kommunikation in WANs oder über das Internet kann VIPP in HTTP getunnelt werden.

Vorbereitungsereignis – Ein Vorbereitungsereignis wird gefeuert, bevor eine Aktion durchgeführt wird. Damit ist es möglich, die Aktion zu prüfen und gegebenenfalls durch eine `VetoException` zu verhindern.

Vorlagentopologie – Die nach Vorlagen angeordnete Struktur von VIP-Objekten innerhalb einer Website (Vorlagenstruktur)

Vorlegen – Workflow-Aktion von VIP ContentManager: Bevor ein neu angelegtes oder geändertes Objekt veröffentlicht werden kann, muss es der Qualitätssicherung zur Prüfung vorgelegt werden. Durch das Vorlegen werden die Änderungen am Objekt in der QS-Sicht sichtbar.

Zugriffssteuerungsliste – Für jedes VIP-Objekt können Benutzer, Gruppen, Rollen und Gruppenrollen festgelegt werden, die Zugriff auf dieses Objekt haben sollen. Die einzelnen Zugriffsrechte werden für jeden Zugriffsberechtigten separat festgelegt. Wird auch als ACL (Access Control List) bezeichnet.

Zurückgeben – Workflow-Aktion von VIP ContentManager: Ein ausgeliehenes und bearbeitetes Objekt wird durch die Aktion “Zurückgeben” an das VIP-CM-System zurückgegeben. Damit werden die vorgenommenen Änderungen in der Edit-Sicht sichtbar. Das Objekt wird wieder mit der Vorlage verknüpft und steht zur weiteren Bearbeitung zur Verfügung.

Index

B

C

D

E

F

G

J

K

L

M

N

O

R

S

T

U

V

W

Z